



**SOFTWARE DOMAIN MODEL
INTEGRATION METHODOLOGY FOR
FORMAL SPECIFICATIONS**

THESIS

Joel C. Nonnweiler, First Lieutenant, USAF

AFIT/GCS/ENG/01M-07

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

20010706 132

REPORT DOCUMENTATION PAGE					Form Approved OMB No. 0704-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</p>						
1. REPORT DATE (DD-MM-YYYY) 20-03-2001		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From - To) Aug 99 - Mar 01		
4. TITLE AND SUBTITLE Software Domain Model Integration Methodology for Formal Specifications				5a. CONTRACT NUMBER		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Joel C. Nonnweiler, 1Lt, USAF				5d. PROJECT NUMBER		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P. Street, Building 640 WPAFB, OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/01M-07		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Robert L. Herklotz AFOSR/NM, Program Manager: Software and Systems 801 N. Randolph St., Room 732 Arlington, VA 22203-1977 (703) 696-6565				10. SPONSOR/MONITOR'S ACRONYM(S)		
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED						
13. SUPPLEMENTARY NOTES Dr. T.C. Hartrum, Associate Professor Emeritus, 937-255-3636 x4581, thomas.hartrum@afit.af.mil						
14. ABSTRACT Using formal methods to create automatic code generation systems is one of the goals of Knowledge Based Software Engineering (KBSE) groups. The research of the Air Force Institute of Technology KBSE group has focused on the utilization of formal languages to represent domain model knowledge within this process. The code generation process centers around correctness preserving transformations that convert domain models from their analysis representations through design to the resulting implementation code. The diversity of the software systems that can be developed in this manner is limited only by the availability of suitable domain models. Therefore it should be possible to combine existing domain models when no single model is able to completely satisfy the requirements by itself. This work proposes a methodology that can be used to integrate domain models represented by formal languages. The integration ensures that the correctness of each input model is maintained while adding the desired functionality to the integrated model. Further, because of the inherent knowledge captured in the domain models, automated tool support can be developed to assist the application engineer in this process.						
15. SUBJECT TERMS Software engineering, Model theory, Integration, Specifications, Methodology						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 176	19a. NAME OF RESPONSIBLE PERSON Dr. Thomas C. Hartrum	
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) 937-255-3636 x4581	

AFIT/GCS/ENG/01M-07

SOFTWARE DOMAIN MODEL INTEGRATION METHODOLOGY FOR
FORMAL SPECIFICATIONS

THESIS

Presented to the Faculty of the Graduate School of Engineering and Management
of the Air Force Institute of Technology
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Joel C. Nonnweiler, B.S.

First Lieutenant, USAF

March 2001

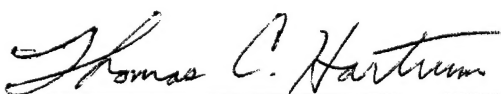
Approved for public release, distribution unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense or the U.S. Government.

SOFTWARE DOMAIN MODEL INTEGRATION METHODOLOGY FOR
FORMAL SPECIFICATIONS

Joel C. Nonnweiler, B.S.
First Lieutenant, USAF

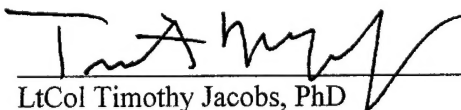
Approved:



Dr. Thomas C. Hartrum
Committee Chairman

20 FEB 01

date



LtCol Timothy Jacobs, PhD
Committee Member

20 FEB 01

date



Maj Scott A. DeLoach, PhD
Committee Member

20 Feb 01

date

Acknowledgments

Before I begin thanking the people, without whom I would have never completed this effort, let me first give thanks to God for giving myself and my family the blessing of this opportunity and the perseverance to endure it even when the going got rough.

I would like to extend a special thank you to Dr. Thomas Hartrum for all of his guidance and patience throughout this process. Our brainstorming sessions and his probing questions continually provided me with the many keys that helped unlock doors and shed light on new challenges within my research. Additionally, I would like to thank each of the members of the Agent Research Working Group: LtCol Timothy Jacobs, Major Scott DeLoach, Capt Athie Self, Lt Clint Sparkman, and Lt Scott O'Malley who provided so much in terms of technical assistance and motivation during our weekly meetings. I owe a debt of gratitude to _____ for helping me transform my sometimes incoherent writing style into a readable work.

Most importantly, I would like to thank my wife _____ and children _____ for their patience and understanding during these past eighteen months.

Joel Carl Nonnweiler

Table of Contents

	Page
Acknowledgments	iv
List of Figures	ix
List of Tables	xiii
Abstract	xiv
1. Introduction	1
1.1. Background (AWSOME)	3
1.2. Problem	6
1.3. Problem Analysis	7
1.4. Scope	9
1.5. Approach	9
1.6. Assumptions	11
1.7. Thesis Overview	11
2. Background	13
2.1. Domain Theory	13
2.1.1. Structural	14
2.1.2. Functional	17
2.1.3. Dynamic	19
2.1.4. Abstract Model (AST)	22
2.2. Well-Formed Domain Models	22
2.2.1. One System Class	23
2.2.2. Unique Identifiers	23
2.2.3. Analysis Type Declarations	23
2.2.4. No Class Attributes	24
2.2.5. Actions on Transitions in the Dynamic Model	24
2.2.6. Functions are Not Actions	25
2.2.7. Restricted Class Visibility	25
2.2.8. Intra-Model Event Associations	26
2.3. Software Component Theory	26
2.3.1. Component Properties	27
2.3.2. Interface Contracts	28
2.4. Type Theory	29
2.4.1. Range Restrictions	30
2.4.2. Type Bridging	32

3. Domain Integration Analysis	34
3.1. Initial Hypothesis	34
3.1.1. Domain Selection	34
3.1.2. Hooks	35
3.1.3. Middleware	36
3.2. Refined Hypothesis	37
3.2.1. Interface Contracts	38
3.2.2. Conversion Trigger Strategies	40
3.2.2.1. Counter-Based Trigger Strategy	40
3.2.2.2. Event-Based Trigger Strategy	41
3.2.2.3. Value-Based Trigger Strategy	42
3.2.2.4. User-Based Trigger Strategy	42
3.2.3. Communication Patterns	42
3.2.3.1. Transfer	43
3.2.3.2. Merge	44
3.2.3.3. Split	44
3.2.3.4. Mixed	45
3.2.3.5. Combined	46
3.2.4. Event Connection Patterns	46
3.2.4.1. Inter-Model Event Connections	47
3.2.4.2. Intra-Model Event Connections	48
3.2.4.3. Severed Intra-Model Event Connections	48
3.2.5. Event Parameter Patterns	49
3.3. Final Solution	50
3.3.1. Domain Model Deconfliction	50
3.3.2. Integrated Model Creation	52
3.3.3. Model Interface Conversion (MIC) Analysis	53
3.3.4. MIC Creation	55
3.3.4.1. MIC Structural Components	55
3.3.4.1.1. Receive Event Parameter Attributes	55
3.3.4.1.2. Trigger Evaluation Attributes	56
3.3.4.2. MIC Functional Components	57
3.3.4.2.1. Conversion Initialization Procedures	57
3.3.4.2.2. Process Receive Event Procedures	57
3.3.4.2.3. Trigger Evaluation Functions	58
3.3.4.2.4. Process Send Event Procedures	58
3.3.4.2.5. Parameter (Send Event) Creation Functions	59
3.3.4.3. MIC Dynamic Components	60

4. Domain Integration Methodology	63
4.1. Integration Analysis	63
4.1.1. Input Model Selection (Pre-Integration Step p1)	64
4.1.2. Communication Pattern Identification (Pre-Integration Step p2)	65
4.1.3. Communication Pattern Analysis (Pre-Integration Step p3)	66
4.2. Generic (UML) Domain Integration Methodology	67
4.2.1. Deconflict the Input Models (Step 1)	68
4.2.2. Create the New Integrated Model (Step 2)	69
4.2.3. Create each MIC's Structural Component (Step 3)	70
4.2.4. Create each MIC's Functional Component (Step 4)	70
4.2.5. Create each MIC's Dynamic Component (Step 5)	71
4.3. AWL Specific Domain Integration Methodology	72
4.3.1. Input Model Selection (Pre-Integration Step p1)	73
4.3.2. Communication Pattern Identification (Pre-Integration Step p2)	73
4.3.3. Communication Pattern Analysis (Pre-Integration Step p3)	74
4.3.4. Deconflict the Input Models (Step 1)	74
4.3.5. Create the New Integrated Model (Step 2)	76
4.3.6. Create each MIC's Structural Component (Step 3)	77
4.3.7. Create each MIC's Functional Component (Step 4)	79
4.3.8. Create each MIC's Dynamic Component (Step 5)	83
4.4. AWSOME Domain Model Integration Tool (ADMIT)	85
5. Domain Integration Methodology Demonstration	87
5.1. Security Manager and Room Manager Demonstration	87
5.1.1. Integration Analysis	88
5.1.1.1. Input Model Selection (Pre-Integration Step p1)	88
5.1.1.2. Communication Pattern Identification (Pre-Integration Step p2)	88
5.1.1.3. Communication Pattern Analysis (Pre-Integration Step p3)	91
5.1.2. Domain Integration Methodology	93
5.1.2.1. Deconflict the Input Models (Step 1)	93
5.1.2.2. Create the New Integrated Model (Step 2)	94
5.1.2.3. Create each MIC's Structural Component (Step 3)	95
5.1.2.4. Create each MIC's Functional Component (Step 4)	96
5.1.2.5. Create each MIC's Dynamic Component (Step 5)	97
5.1.3. ADMIT Demonstration	99
5.2. Additional Examples (non-simple communication patterns and conversions)	103
5.2.1. Merge Communication Pattern (multiple attribute values)	103
5.2.2. Combined Communication Pattern	105
5.2.3. Split Event Pattern	107

5.3.	Multi-Agent Domain Model Integration	108
5.3.1.	Room Manager and agentMom Integration Demonstration	108
5.3.2.	Room Manager and Channel Integration Demonstration	110
6.	Results, Conclusions, and Recommendations	114
6.1.	Results	114
6.2.	Conclusions	114
6.3.	Synopsis	115
6.4.	Recommendations for Future Research	116
6.4.1.	Alternate Container Types (AWL Methodology)	116
6.4.2.	Information Retrieval Techniques in Domain Analysis	117
6.4.3.	Artificial Intelligence Techniques in Domain Selection	117
6.4.4.	Verifying Correctness of Supplied Expressions	118
6.5.	Summary	118
Appendix A.	AWSOME Wide Spectrum Language (AWL) Quick Reference	119
Appendix B.	AWSOME's WsClasses AST Structure	124
Appendix C.	Room Manager System Domain Model (AWL)	125
Appendix D.	Security Manager System Domain Model (AWL)	131
Appendix E.	Channel Domain Model (AWL)	147
Appendix F.	Configuration Management for ADMIT	157
Bibliography	158
Vita	160

List of Figures

	Page
Figure 1. Formal Approach to the Creation of Correct Domain Specific Software.....	3
Figure 2. Modified Formal Specification Generation Process.....	4
Figure 3. Structural Domain Model Example - UML.....	15
Figure 4. Structural Domain Model Example - AWL.....	16
Figure 5. Functional Domain Model Example - AWL	18
Figure 6. Dynamic Domain Model - UML	19
Figure 7. Dynamic Domain Model Example - UML.....	20
Figure 8. Dynamic Domain Model Example - AWL.....	21
Figure 9. Domain Model Interface Contract Example.....	29
Figure 10. Source-to-Target Range Categories.....	30
Figure 11. Range Restriction Examples.....	31
Figure 12. Counter-, Event-, and Value-Based Trigger Strategies	41
Figure 13. Transfer Communication Pattern.....	43
Figure 14. Merge Communication Pattern.....	44
Figure 15. Split Communication Pattern.....	45
Figure 16. Mixed Communication Pattern.....	45
Figure 17. Combined Communication Pattern	46
Figure 18. Inter-Model Event Connections.....	47
Figure 19. Intra-Model Event Connections.....	48
Figure 20. Severed Intra-Model Event Connection	49
Figure 21. Identifying Communication Patterns (All-encompassing vs. Minimal).....	54

Figure 22. Input Model's Send/Receive Events become the MIC's Receive/Send Events	54
Figure 23. Convert Function's Assignment Expression Creation Flowchart	59
Figure 24. MIC's Dynamic Model State Diagram (Generic)	61
Figure 25. Sample Integration Analysis Worksheet (Input Model Selection)	64
Figure 26. Sample Integration Analysis Worksheet (Communication Pattern Identification)	65
Figure 27. Sample Integration Analysis Worksheet (Communication Pattern Analysis)	66
Figure 28. Domain Model Integration Methodology	67
Figure 29. Input Models 1 and 2 Prior to Identifier Deconfliction	68
Figure 30. Input Models 1 and 2 Post-Deconfliction	69
Figure 31. New Integrated Model (input models: Model1 and Model2)	69
Figure 32. MIC Created for a Communication Pattern Example	70
Figure 33. Example MIC's Functional Component from a sample communication pattern	71
Figure 34. Example MIC's Dynamic Model	72
Figure 35. The ADMIT Object Model	85
Figure 36. Secure Room Manager Input Model Selection Worksheet	89
Figure 37. The Integration Plan for the Room and Security Models	90
Figure 38. Secure Room Manager Communication Pattern Identification Worksheet	91
Figure 39. Secure Room Manager Communication Pattern Analysis Worksheet	93
Figure 40. Integrated Secure Room Manager Aggregate Domain Model	95
Figure 41. Secure Room Manager MIC (without the dynamic model)	97
Figure 42. Dynamic Model for Secure Room Manager's MIC	98
Figure 43. MIC1's Dynamic Model (Secure Room Manager)	98
Figure 44. Pre-step 1: Input Model Selection (<i>and verification</i>)	100
Figure 45. Pre-step 2: Communication Pattern Identification	100
Figure 46. Pre-Step 3: Communication Pattern Analysis (<i>pattern recognition</i>)	101
Figure 47. Pre-step 3: Communication Pattern Analysis (<i>event selection</i>)	101

Figure 48. Pre-step 3: Communication Pattern Analysis (<i>conversion expression</i>).....	102
Figure 49. Automated Integrated Model and MIC Creation Steps	102
Figure 50. Merge Communication Pattern Example.....	104
Figure 51. MIC Generated from Merge Communication Pattern Example.....	104
Figure 52. Combined Communication Pattern Example	105
Figure 53. Combined Communication Pattern Example (Dynamic Model).....	105
Figure 54. MIC Generated from Section 5.2.2's Example	106
Figure 55. Split Event Pattern Example.....	107
Figure 56. Integrated Model Associations Generated by a Split Event Pattern	107
Figure 57. agentMom Object Model [14, 16]	108
Figure 58. Multi-Agent System Creation Process Using agentMom.....	109
Figure 59. Room Manager System Message Passing Diagram (only two messages).....	109
Figure 60. Multi-agent Room Manager System Implemented using agentMom.....	110
Figure 61. Channel Aggregate Model.....	111
Figure 62. Channel Message Passing Diagram.....	111
Figure 63. Room Manager / Channel Integration Analysis	112
Figure 64. ADMIT Screen Capture (Integration of Room Manager and Channel)	113
Figure 65. AWSOME's WsClasses AST Inheritance Diagram	124
Figure 66. Room Manager System Aggregate Domain Model.....	125
Figure 67. Room Manager System Association Model	125
Figure 68. Room Manager's RoomUser State Diagram.....	126
Figure 69. Room Manager's RoomKeeper State Diagram.....	126
Figure 70. Security Manager's Aggregate Domain Model	131
Figure 71. Security Manager's SecSysUI State Diagram.....	132
Figure 72. Security Manager's AdmMgr State Diagram.....	132
Figure 73. Security Manager's AccMgr State Diagram	133

Figure 74. Security Manager's AppMgr State Diagram..... 133

Figure 75. Channel Aggregate Domain Model 147

Figure 76. Channel's Client and Server Dynamic Models 147

List of Tables

	Page
Table 1. Valid and Invalid Type Casting Examples (based on Figure 11)	31
Table 2. Type Mapping and Type Casting Definitions.....	32
Table 3. Type Bridging Issues (Combination of Type Conversions Allowed).....	33
Table 4. Example Model Interface Contract (Model 1 from Figure 9).....	38
Table 5. Example Model Interface Contract (Model 2 from Figure 9).....	39
Table 6. Secure Room Manger's Updated Interface Contract (only modified entries).....	99

Abstract

Using formal methods to create automatic code generation systems is one of the goals of Knowledge Based Software Engineering (KBSE) groups. The research of the Air Force Institute of Technology KBSE group has focused on the utilization of formal languages to represent domain model knowledge within this process. The code generation process centers around correctness preserving transformations that convert domain models from their analysis representations through design to the resulting implementation code. The diversity of the software systems that can be developed in this manner is limited only by the availability of suitable domain models. Therefore it should be possible to combine existing domain models when no single model is able to completely satisfy the requirements by itself. This work proposes a methodology that can be used to integrate domain models represented by formal languages. The integration ensures that the correctness of each input model is maintained while adding the desired functionality to the integrated model. Further, because of the inherent knowledge captured in the domain models, automated tool support can be developed to assist the application engineer in this process.

SOFTWARE DOMAIN MODEL INTEGRATION METHODOLOGY FOR FORMAL SPECIFICATIONS

1. Introduction

Consider the following example: A user has a requirement for a battlefield simulation system that will allow commanders to evaluate the performance of new aircraft. The user produces a software requirement document that defines the problem statement and delivers it to a software application engineer. Suppose that the engineer has access to an existing domain model of a battlefield simulator. With such a model, a formal specification for the battlefield simulator system could easily be derived. Likewise, formal specifications could be created for each requested aircraft that had a formally defined domain model. Given that all of the necessary domain models are available, the application engineer could integrate them into one comprehensive model. With the newly integrated model, a formal specification for the requested system could be generated which could then be used in an automated process to generate the code for the requested system.

Sound like a dream come true? It is closer to reality than may be imagined. There has been extensive research into the automated process of generating code using correctness preserving transformations over a formal specification [5, 10, 11, 17-19]. The challenge of the preceding scenario is that of the domain integration. While domain model knowledge is available and relatively well understood, it must be formally defined for use in both the integration and harvesting processes. By understanding and defining the process of integration for domain models, the software engineering principle of software reuse becomes more attainable. As the library of formally defined domain models increases, the opportunity to use provably correct domains to satisfy software system requirements will also grow. For example, while generating the specification for a jet aircraft system, the requirements for a fuel and timing subsystem are

identified. The application engineer, while working from the model of the jet provided by the domain expert, searches the available sources of domains to locate fuel and timing system models. Rather than having to produce models of the deficient systems, a time consuming, expensive, and difficult task, domain model integration provides the opportunity to utilize existing models that were created for other reasons.

Domain Integration (DI) uses a building block approach to software system creation. This technology greatly reduces the amount of effort necessary to create new systems by reducing original code to those portions that are not already found in other domain models. In addition to integrating complete domain models, it is possible to integrate portions of models by first extracting pertinent subsystems from a domain. This complimentary technology, Elicitor-Harvester (EH), adds a new dimension of flexibility to the application engineer with regard to the system models that can be created by utilizing existing model libraries. EH is simply the process of selecting some meaningful subsystem from a given model that satisfies in some part the requirements of a given problem statement. The subsystems gathered from existing domain models can then be used as input for the DI process in order to create a new integrated model with the required functionality. While this provides a further dimension of software reuse, it also illustrates the need to protect the application engineer from becoming a domain expert for every possible useful domain model. If a detailed understanding of each domain were required for either the DI or EH process, the benefits of model reuse would be diminished, if not completely nullified. To this end, a methodology is required that shields the engineer from becoming bogged-down with the burden of learning the low-level details of each involved domain. The methodology must provide a systematic approach to the task of integrating domain models. Additionally, the creation of a formal methodology leads to automation or semi-automation of the integration process, which will enhance the existing Knowledge Based Software Engineering (KBSE) formal system creation process.

1.1. Background (AWSOME)

There has been considerable research within AFIT's KBSE group concerning the process that transforms knowledge of a domain model over a problem space into correct software solutions [15-19]. The AFIT Wide-Spectrum Object-Oriented Modeling Environment (AWSOME) is an environment in which these transforms can take place [16, 18]. The process involves several major transforming steps, but uses the same language throughout to abstractly represent the system until executable code is produced. The following discussion briefly outlines the process of transforms within AWSOME.

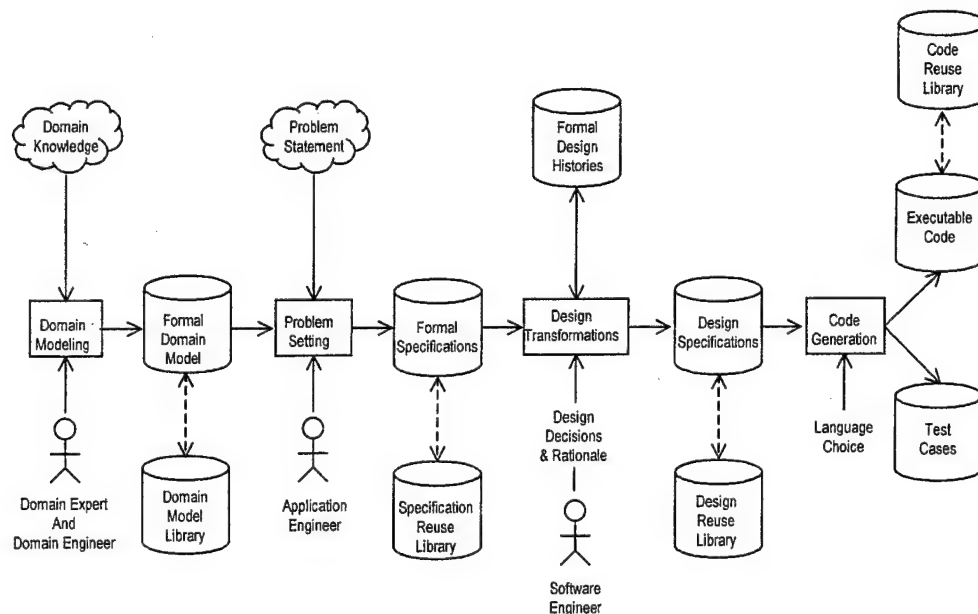


Figure 1. Formal Approach to the Creation of Correct Domain Specific Software

The first stage, *Domain Modeling*, involves domain experts and engineers using their extensive knowledge of their respective domains to create model representations of the systems. The models, which are represented by a formal surface syntax language such as Z, COIL, AWL, etc. are parsed into abstract syntax trees (AST). ASTs are the in-memory hierarchical computer representations of the model, and parsing converts the formal surface syntax to the abstract form.

As parsers are not available for all model representation languages, other methods exist by which ASTs are created, but such methods are not discussed here. AWSOME uses this abstract representation of the knowledge throughout the remaining transforms. The output of domain modeling is the *Formal Domain Model*, denoted DOM, which can be saved in libraries for future utilization in both the surface syntax or AST formats, and is the starting point for the next stage in the process. The next stage is of particular interest to this research effort as it deals with creating formal specifications from the input models. Application engineers use the *Problem Setting* process to create system *Formal Specifications* by satisfying the *Problem Statement* over a given domain model.

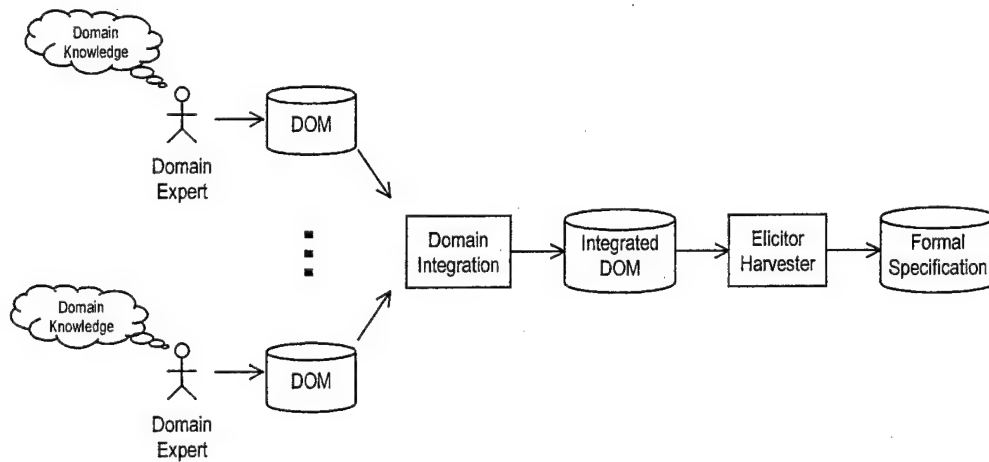


Figure 2. Modified Formal Specification Generation Process

The formal specification generation process is the subject of on-going research regarding the use of EH techniques [5, 16, 17, 19] to form system specifications from domains in a semi-automated or tool-assisted fashion. Likewise, this stage is home to the proposed integration process to assist the application engineer in the task of specification creation. Notice in Figure 2, the application engineer now has access to multiple DOMs with which to generate the formal specification. Like the domain models, the formal specifications can also be stored for future use in software libraries. It is the generation of these formal specifications that is the focus of this

research. The specification that is the result of this stage is then subjected to *Design Transforms* that convert the system from an analysis to a design representation. At this stage, software engineers make design decisions based on knowledge of software architectures, problem specifications, and other concerns. The formal design histories are captured and stored for future use if code regeneration is required. The final stage, *Code Generation*, converts the transformed AST into some chosen language [15].

Central to the code generation process is the domain. It is the piece of information that exists prior to the problem statement and remains after new systems are created. Domain knowledge is the collection of information that describes selected aspects of a given system. It is from this knowledge that domain experts create domain models for use in creating formal specifications. It is important to note that while domain models formally describe systems, they may be incomplete with respect to the desired system specification. Domain experts may or may not be aware of the many possible applications that could utilize their models. Thus, application engineers may need to add to or modify available models to fit their problem space before system specification can be completed. However, it is also possible to select all or parts of other models to supply the missing system components needed in the generation of a complete specification.

Additionally, there has been ongoing work with utilizing EH techniques as a semi-automated support for the creation of the formal specification AST over a problem statement using an input DOM. Introducing integrated DOMs to EH can enhance this effort. However, many questions remain concerning the integration methodologies that would ensure that the correctness of DOMs are maintained throughout the AST integration process. Furthermore, the ability to introduce multiple DOMs into the formal specification generation process could significantly aid system engineers in the creation of new software systems. Entire existing systems or parts of such systems could be reused in the generation of other formal specifications. This implies the use of libraries of categorized models that would support many different

functions such as multi-agent frameworks, system security protocols, or simulator systems to name a few.

1.2. Problem

“Develop an integration methodology for combining formal domain models to achieve the desired functionality while retaining the correctness of the input models”

In order to successfully realize the goal of this research, several key concepts needed to be addressed. These concepts were discovered during both the background study and the development of the actual methodology. The first challenge revolved around the selection of domain models to be used in the integration process. The actual selection of each model was not addressed specifically, other than to assert the assumption that the input models are appropriate for the integration. However, once selected, the models must be verified to ensure that they conform to a standard well-defined format. The real challenge for the research was to discover a method by which the models could be integrated without the application engineer having an in-depth knowledge of the input models. Building upon the knowledge of why and how to integrate models, the research then had to focus on identifying the connecting communication(s) between the input models, and how to use them to create a sort of “middleware” that would facilitate the connection. The problem of creating a standard methodology for the creation of this connecting specification focused into two areas. First, the topic of converting values and types, including type range restrictions had to be considered. Second, the flow of processing, including the required functions, for the middleware had to be defined. In this manner, the methodology could ensure that the resulting integrated model would be both verifiably correct and contain the desired functionality provided by the merging of the input domain models.

1.3. Problem Analysis

The solution to this problem was approached by following two paths. First, the methodology was built around the existing environment provided at AFIT, i.e. AWSOME. This was done by generating real-world examples using the surface syntax AWL which could then be parsed into the AWSOME environment and used to perform actual integrations. However, the end result of this approach led to a methodology specific to this one environment. What was needed was a generalized solution that could be applied to any specific implementation as needed. Thus for the second approach, the Unified Modeling Language (UML) [9] was used to provide a basis for the general methodology. A mapping between the two approaches allows all steps present in the AWL implementation to be traced to their generalized steps in the UML process. The resulting solution to the stated problem is as follows:

1. Select appropriate input domain models.
2. Identify connecting communication patterns through the use of interface contracts.
3. Create the new model, and copy the input models into it.
4. Design a Model Interface Conversion (MIC) for each identified pattern, addressing type and value conversion of shared information and execution strategies.

In order to verify that the proposed methodology is in fact a valid solution to the stated problem, two input domain models were integrated using the methodology. After the newly integrated model was completed, it was inspected to verify correctness. While a complete demonstration is provided to be used as an example, portions of “contrived” models are shown to illustrate difficult or interesting aspects of this study. Additionally, as domain models are of the utmost importance to this project, special care is taken in defining what a “well-formed” domain model is. Only after models are verified and determined to be well-formed using the rules established for the AWL surface syntax, can the integration proceed.

In dealing with the task of identifying the connecting communication patterns, the idea of software components became very appealing. One of the early rules that was established was that input models could not be modified and still retain their "verified correct" status. Thus the input models were treated as black-boxed components. An added benefit to this analogy was the concept of using events that reached outside of models (inter-model) as the component's interface contract. If domain model documentation could focus on stating exactly what parameters in outgoing events were guaranteed to be and what parameters in incoming events required, then the application engineer would have a quick and concise map to determine what interface conversions were required.

The final focus of the methodology is the generation of the Model Interface Conversions (MICs) which model the application engineer's design of the connecting communication. There are three major areas of concern in the MIC generation process. The first area has to do with the MIC's incoming events. Specifically, the MIC must somehow store value(s) contained in the incoming events' parameters to be used later in the generation of the outgoing events. The second area of concern is determining how the outgoing event parameters are correctly generated. This challenge was met by exploring the topic of type conversions. Specifically, rules for converting values between different types were examined, and steps were developed that ensure range restrictions between input and output variables are satisfied. Thus for each outgoing event parameter, a unique conversion function is required to be called from a master conversion procedure. The third area of concern deals with the MIC's dynamic model. A method of determining when the MIC is ready to generate the outgoing events had to be developed. Because each inter-model communication pattern is unique, a standard approach was needed. The determining method evolved into the concept of categories of trigger strategies, from which application engineers can choose to meet the demands of each interface communication pattern.

1.4. Scope

During this research there arose a number of issues that, while important to point out and explain, were not specifically addressed within the scope of this methodology. Both the storing and selection of domain models are beyond the scope of this thesis. The topic of creating and maintaining domain libraries is not explored, but references are made about searching through them. Additionally, it is beyond the scope of this study to make the connection between choosing appropriate domain models and the presented problem statement.

Another area that was beyond the scope of this problem was the verification of the conversion logic. The logic, discussed in detail in later chapters, is supplied by the application engineer in the creation of parameters for the outgoing events. While a decision flowchart is provided to assist the engineer in creating correct expressions and there is detailed discussion of both type bridging and range restrictions for comparing source and target variables, the actual verification of these expressions is left to the discretion of the engineer.

1.5. Approach

To meet the proposed research objectives, the following approach was followed:

1. *Become familiar with the current KBSE tools.* The example models used to develop the methodology were created using the AWL wide spectrum language. Additionally, the AWSOME environment was used to demonstrate the feasibility of automating the integration methodology.
2. *Study domain modeling topics.* In order to create the domain integration methodology, the aggregated domain model structure needed to be formally defined. Both the UML and AWL representations of domain models were studied. This allowed the standardization of the methodology.

3. *Create a cache of DOMs to be used in developing the methodology.* A library of AWL domain models that adhere to the well-formed domain rules needed to be created. Previously created models had to be converted from other formal languages, such as Z and COIL, in order to have domains with which to integrate. Additionally, a security system domain model was created to demonstrate the ability to provide security to existing systems that previously did not have the protection of a security system.
4. *Manually integrate available DOMs by developing an integration methodology.* Hand integration of both UML and AWL domain models was accomplished in order to understand the steps required to successfully combine models. These integrations provided insight into the many topics of concern that were addressed in creating the methodology.
5. *Study the AFIT Multi-Agent Environment (agentMom).* This background study was required to facilitate the integration of agent-based and non-agent oriented domain models.
6. *Design and implement the Integration Tool.* Knowledge gained from previous research was used to develop a semi-automated tool that demonstrated the feasibility of automating the methodology. Additionally, the tool provides AWSOME with increased functionality during the stage of formal specification generation. The completed formal specification generation package allows a closer integration between AWSOME and AgentTool by providing a method for integrating multi-agent DOMs with other formally specified DOMs.
7. *Test the Integration Tool.* The tool was used to assist in the integration of domains during the testing and evaluation stage of the methodology development. The verification of the tool was accomplished by comparing integrated models created by both a hand and tool assisted generation.

8. *Analyze results of the methodology.* After the methodology was created and verified, an analysis was performed to determine how successful the effort was. The analysis was useful in identifying alternate approaches and areas of possible future research.

1.6. Assumptions

For whatever reasons the application engineer deems necessary, it is assumed that the selected input domain models are both relevant to the problem statement and have suitable interfaces that facilitate a successful integration. Another assumption relied on is that possible domain models have adequate documentation that specifically allows application engineers to quickly identify published interface contracts. These contracts inform the integrator what information, in the form of event parameters, is provided or required to successfully use the model in conjunction with other models or systems. As is discussed in Chapter 3, the contracts should be the only part of the input model that an engineer would have to become familiar with.

1.7. Thesis Overview

Chapter 1, Introduction, is intended to give the reader a sense of the void or problem which this research will satisfy. It introduces the concept of what domain integration is and why it is beneficial to the field of software engineering. Chapter 2, Background, provides the reader with an understanding of domain models. Specifically, background is provided on the UML, AWT, and AST model representations. Additionally, topics such as viewing models as software components and how interface contracts appear in models are addressed. Finally, this chapter provides background on the topic of type bridging as related to type conversions and range restrictions. Chapter 3, Domain Integration Analysis, focuses on the development process of creating the integration methodology. It includes topics such as analyzing communication

patterns, event connection patterns, and event parameter patterns, and culminates with the development of the Model Interface Conversion (MIC) concept. Chapter 4, Domain Integration Methodology, presents the methodology in four phases. First, the pre-integration analysis steps are described. Second, the generic UML methodology is presented followed by the AWL language specific version. Lastly, the demonstration integration support tool, AWSOME Domain Model Integration Tool (ADMIT) is briefly discussed. Chapter 5, AWL Domain Integration Methodology Demonstration, walks the reader through an actual integration between the provided Room Manager System model and the Security Manager System model. Additional examples are provided to demonstrate the ability to handle other possible combinations of communication and event connection patterns. Additionally, an integration with a multi-agent system is attempted to discuss the issues of integrating with models that do not fit the well-form domain model definition. Chapter 6, Results, Conclusions, and Recommendations, addresses the conclusions of the research as well as outlining possible areas for future research.

2. Background

In order to develop a methodology that will successfully guide software engineers through the process of integrating input domain models, the following topics of interest must be visited. First, domain theory and its structural, functional, and dynamic components must be understood, including its graphical (UML), surface syntax (AWL), and computer memory (AST) representations. Second, domain models must be examined in the light of software component theory, specifically the concepts of black-box and interface contracts. Finally, type conversions and value computations are explored, including a discussion on range limitations with regard to source and target variables. These discussions are important in that they lay the foundation for decisions and analysis in the integration techniques.

2.1. Domain Theory

Domain models are relatively well understood. The domain object model (DOM) is a collection of closely related object classes that are bound through relationships, and represent a system that has a well defined structure and behavior [1, 5, 16, 17, 19]. Domains are defined using the object-oriented paradigm, and as such, the models' classes are composed of attributes and methods. The models are created by domain experts who construct classes and combine them in various patterns or architectures that describe the system. While it is said that domain experts create the model, it is more likely that domain experts employ software engineers to create the formal specifications. The domain experts communicate the properties of the model to the engineers, who in turn build the formal model. Validation that the model is completely accurate is of utmost importance at this stage.

Expressing what is desired in the model can be done in a number of ways. One method of depicting a domain is via the Unified Modeling Language (UML) [9], though others exist such

as the Object Modeling Technique (OMT) [13]. While using these graphically based methods of domain representation allows easy communication of the overall intent of the model, a formal method must be used that excludes the possibility of ambiguous meaning. For the purposes of this research, the surface syntax of AWL [10, 11, 16, 18] is used. Additionally, using a formal language provides the opportunity for a parser to translate the unambiguous model definition into a form which the computer can manipulate, in this case an abstract syntax tree (AST) [5].

Models are defined by three components [4, 12]: structural (units and associations), functional (subprograms – methods or operations), and dynamic (events and state transitions). Each of these model components is discussed below. Each topic is first illustrated by the UML representation and then by the AWL representation. After the three components are explored, the AST representation is addressed to allow a better understanding of how the AWL model is stored in memory and how manipulations to the tree can affect the desired results.

2.1.1. Structural. The structural component is the basis of the DOM, and is a mixture of classes and relationships. The example in Figure 3 depicts a simple sample model that describes a generic sports team. The model consists of four objects: Team, Player, Game, and Quarter, which extends Game, and two relationships “*has*” and “*playedIn*”. The class Player is expanded to show its composition: Class name (Player), Attributes (name and number), and Methods (setName and getNumb). The relationships between the individual classes form the bonds that glue the independent classes into a cohesive model. Each relationship is composed of a relationship name, roles for the participating classes, and the multiplicity of each participant. There are two types of relationships within the domain model, and each has its own purpose.

The association, shown in Figure 3 as “*playedIn*”, implies a relationship between the classes Player and Quarter. In this case, the meaning is that Player(s) plays in Quarter(s), and that Player has the role of “*person*” and Quarter has the role of “*gameQtr*”. Additionally, the multiplicity (how many objects of the specified class are required or allowed in the relationship)

designated for each role is zero to many “0..*”. In the previous example, “*playedIn*” is a binary relationship between two classes. However, relationships can exist between more than two classes. These relationships are equally valid and fulfill an important role in the object model. Aggregation, on the other hand, is a specialized form of an association. Unlike an association, an aggregation’s roles are fixed to “parent” and “child”. In this example, Team (parent) “*hasPlayer*” (relationship name) zero or more Players (child). This unique relationship allows parent classes to have visibility into child classes. Other than this exception, classes are created independent of any other class, having no visibility into other classes. Inheritance is another relationship referenced in the example. However, it is not a relationship in the sense that it joins different classes. Rather, inheritance indicates an extension of a superclass. Quarter inherits all attributes and methods of Game. In effect, Quarter is only an extension of Game, and thus represents only one class when instantiated.

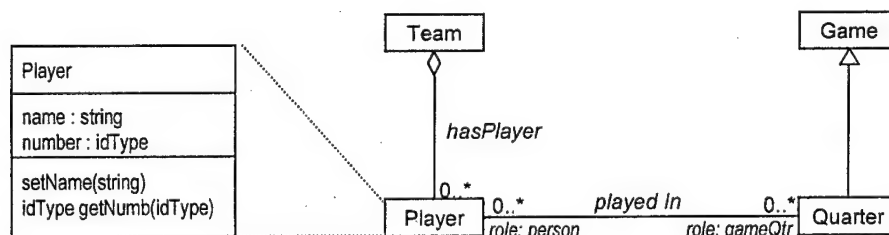


Figure 3. Structural Domain Model Example - UML

An important aspect of class attributes is their definition or declaration based on specific types. In the UML examples, a detailed description of type definitions is not necessary as matching type names are assumed to be compatible, and perhaps more importantly, the internal definition of class methods are not defined. Thus it becomes less important to formally define the class’s attribute types. The same is also true for the parameters identified in the method declarations.

```

package SportsTeam is
  type CHAR is abstract;
  type CHARACTERS is set of CHAR;
  type STRING is sequence of CHARACTERS;
  type IdType is range 10 .. 99;
  type One is range 1 .. 1;
  type ZeroOrMore is range 0 .. *;

  class Team is
  end class;

  class Player is
    name : STRING;
    number : IdType;
    procedure setName(inName : in STRING)
    procedure getNumb(outNumb : out IdType)
  end class;

  class Game is
  end class;

  class Quarter is Game with
  end class;

  association PlayedIn is
    role person : Player multiplicity ZeroOrMore;
    role gameQtr : Quarter multiplicity ZeroOrMore;
  end association

  aggregation HasPlayer is
    parent p : Team multiplicity One;
    child c : Player multiplicity ZeroOrMore;
  end aggregation;
end package;

```

Figure 4. Structural Domain Model Example - AWL

UML quickly communicates the model but at a high level of abstraction. In order to achieve a greater level of detail, and to facilitate the entry of the model into the AST form, a more detailed surface syntax is used. The same SportsTeam example is depicted in Figure 4, but this time the representation is not graphical. The same objects appear (Team, Player, Quarter, and Game) as well as the relationships (PlayedIn and HasPlayer). The classes have the same attributes (name and number) and methods (setName and getNumb). Notice, however, that the detail of the type declarations has been increased to define the types STRING and IdType.

The three domain model aspects (structural, functional, and dynamic) are dependent upon each other. For example, method signatures are an essential part of the structural model, while their internal composition falls within the purview of the model's functional component.

2.1.2. Functional. The definition of the functional model is not specifically addressed by UML. However UML case tools, like Rational Rose, typically provide a way to add the method definitions. The “functionality” of a model is supplied by the pre- and post-conditions as defined in each of the class’s methods [11, 15]. Within the functional aspect of the model, there are two types of methods. One is the procedure, which defines the actions each class can execute. The other is the function, which supplies each class and the entire model the opportunity to simplify constraint expressions wherever they appear. The procedure is the only way in which a class’s attributes can be accessed and/or modified. Functions, on the other hand, cannot be used directly to access class attributes because they cannot be invoked as actions in the class’s dynamic model. Method pre- and post-conditions, in conjunction with class and relationship invariants, provide the definition of the domain’s functional model. These conditions and invariants are composed of boolean expressions, in the form of constraints, that express the behavior of the object. Automatic code generation, an ongoing research focus at AFIT, utilizes these expressions to create provably correct code [11, 15]. While UML offers little in the way of formally representing the functional model, AWL offers a formal syntax to define method pre- and post-conditions.

As is expressed in Figure 5, the post-condition of the method setName guarantees that the local variable “name” will equal the incoming parameter “inName”. Likewise, the method getNumb guarantees the outgoing parameter “outNumb” will have the value of the local attribute “number”. The two methods provide the Player class with the functionality to set the Player’s name and also to get the Player’s number. The post-condition (guarantees) states clearly what will be true after the method has been executed [15]. Similarly, the pre-condition (assumes) clearly states what conditions must be true in order for the method to produce expected results.

```

package SportsTeam is
...
  class Player is
    name : STRING;
    number : IdType;

    procedure setName(inName : in STRING)
      guarantees (name' = inName)

    procedure getNumb(outNumb : out IdType)
      guarantees (outNumb = number)

  end class;
...
end package;

```

Figure 5. Functional Domain Model Example - AWL

Another topic with regard to the functional model's conditions and invariants is the measurement of how inclusive they are. The less restrictive a condition is the more weak it is considered. Alternately, a more restrictive condition implies a stronger expression. The weakest condition is the expression: "True" (completely non-restrictive), and the strongest condition is the expression: "False" (totally restrictive). In AWL, a pre- or post-condition of True is implied by omitting it [10, 11]. Typically, it is preferred that methods have weak preconditions and have strong postconditions. If this is the case, the method can be called in any situation, and would provide an expected result.

As a side note, while constraint invariants over post-conditions in conjunction with the established subprogram declarations describe the functionality of domain models, invariants can be defined over other model entities such as packages (AWL concept), classes, events, and associations. Association invariants play an important role because they provide the means to formally define the membership in model relationships. Note, classes have no knowledge of other classes, hence they cannot reference any other class in order to define relationship membership. Therefore, there needs to be some other means of expressing a cross-class constraint. It is this requirement that the association invariant fulfills.

2.1.3. Dynamic. The dynamic component of a domain model allows its classes to exercise their functional models. The dynamic model is important because it provides the vehicle that allows each of the model's classes to directly control its own attributes. Attributes can only be modified in actions, defined as the class's methods, and actions can only be performed on transitions between states in the class's dynamic model. This is a state-based system that relies on events to trigger transitions [4, 9]. Finally, it is the definition of the transitions that gives the control to the class, which answers the question of when actions are performed.

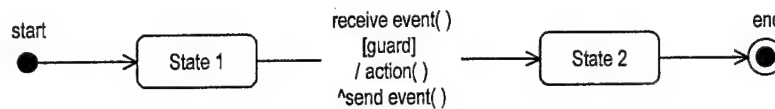


Figure 6. Dynamic Domain Model - UML

Figure 6 depicts the components of the UML representation of the dynamic model. The model is composed of states and transitions. States exist to define “where” the class is at any point, illustrating that a class can only be in one state at any given moment. Correspondingly, depending on the current state, only specified events can be received or sent. Thus being in a certain state will control what actions can be performed at that stage. A special case is the (AUTO)MATIC event which is not really an event in so much as it is a signal for the immediate transition from one state to another. When an appropriate event is received, a guard condition can be applied to further control the execution of an action or the transfer to another state. The guard condition is a boolean expression that utilizes the class's attributes and/or the incoming event's parameter values to determine whether the transition is appropriate to execute.

There are two other points that are important to mention. First, because classes can only exist in one state at any time, only one transition can be executed at any time to leave a given state. If this rule was not enforced, then it would be impossible to determine the class's state

because of the ambiguity of what transition was taken. Secondly, because models use the dynamic model to share information between classes and other models, the parameters of the incoming and outgoing events are the only way to pass attribute values.

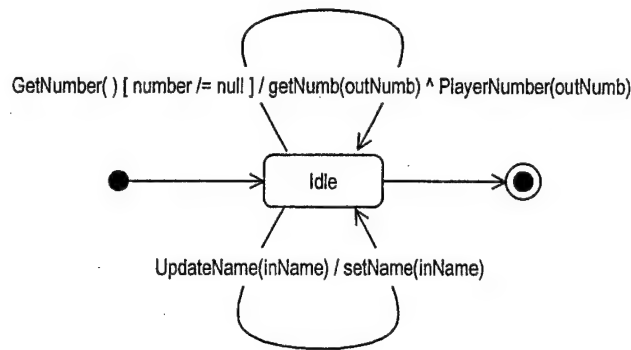


Figure 7. Dynamic Domain Model Example - UML

Continuing the previous examples, Figure 7 demonstrates what the dynamic model for the Player class looks like. The example shows that the only way for the Player's name to change would be if the class was in the Idle state and it received an "UpdateName()" event. If this were the case, then the method/action "setName()" would be executed and the parameter "inName" would be stored in the local attribute "name", as defined in the functional model. Likewise, if the event "GetNumber()" was received while in the Idle state and the local attribute "number" was not null (this is a guard condition), then the action "getNumb()" would be performed. As indicated in both the dynamic and functional models, the value in the local attribute "number" would be stored in the parameter "outNumb" which in turn would be included in the send event "PlayerNumber()".

The corresponding AWL code is found in Figure 8. In it, the dynamic model has been added to the class definition. It contains three major sections. The first is a list of all of the events found in the state diagram. As an aside, the parameters in the receive events are indicated with an "in" while the send event parameters are indicated with an "out". The second section of the dynamic model is the list of all of the class's states. The lists of both the events and states are

used in the third section, which is the transition table. The table describes all of the transitions as graphically depicted in the UML state diagram [10, 11]. Each transition has six components, but only three are required (in *state*, on *receive event*, and to *state*). Only transitions that require a guard condition need an if statement. Likewise, not all transitions perform actions or generate send events.

```

package SportsTeam is

...
class Player is
  name : STRING;
  number : IdType;

  procedure setName(inName : in STRING)
    guarantees (name' = inName)

  procedure getNumb(outNumb : out IdType)
    guarantees (outNumb = number)

  dynamic model is
    event AUTO( );
    event TERMINATE( );
    event GetNumber( );
    event PlayerNumber(outNumb : out IdType);
    event UpdateName(inName : in STRING);

    state START;
    state Idle;
    state STOP;

    transition table is
      in START on AUTO to Idle;
      in Idle on UpdateName do setName to Idle;
      in Idle on GetNumber if number /= null
        do getNumb send PlayerNumber to Idle;
      in Idle on TERMINATE to STOP;
    end transition table;
  end dynamic model;
end class;
...
end package;

```

Figure 8. Dynamic Domain Model Example - AWL

While the UML state diagram is perhaps easier to follow, it is the AWL form that will ultimately be created and parsed into an AST for entry into the transformation process which in turn will convert the formal specification into code. Of special importance is the dynamic

model's events and their parameters. As discussed in Chapter 4, these items will become part of the domain model's interface contracts.

2.1.4. Abstract Model (AST). During the discussion of the domain's structural, functional, and dynamic models, how the model was represented in terms of both UML and AWL is addressed. This section is included to take the domain representation one step further. As an AWL model can be derived from a UML model, an AST can be derived from an AWL file, or any other wide-spectrum formal surface-syntax language. The formal language is parsed into an object-oriented hierarchical tree structure which can be traversed to access or modify objects within the tree [5, 16-19]. A copy of the AWL quick reference guide [10] is provided in Appendix A to assist with the understanding of the examples throughout this paper. Also, Appendix B outlines the WsClasses AST structure, the AWSOME environment, and is included to assist in understanding the transformations required to automate the domain integration process.

2.2. Well-Formed Domain Models

The previous sections examined how domain models are represented in their UML, AWL, and AST forms. Defining what constitutes a "well-formed" domain model is the focus of this section. The rules outlined relate primarily to the AWL representation. UML hides much of the specific concerns relating to the model definition, as this representation is focused on the generic analysis of models. Likewise, the AST representation is nothing more than an internal memory representation of the surface syntax of the model. Thus, these rules are provided as a way of specifying the constraints under which AWL domain models can be integrated using this methodology.

2.2.1. One System Class. Because the AWL formal domain model is structurally made up of classes in aggregate relationships, there must be only one hierarchical tree structure. Well-formed domain models have one root class, called its system class or “top-level” parent, meaning that all other classes can trace a line of ancestors back to the system class. Remember that classes that extend a super class, in an inheritance relationship, trace their line of ancestors through their super class. The one system class rule is required for several reasons. First, it provides a convenient way to handle an entire model in its AST form. Secondly, having a top-level parent class provides the opportunity for the application engineer to address global invariants for the model’s relationships, both aggregate and associations.

2.2.2. Unique Identifiers. The rule of unique identifiers states that all identifiers within the same scope must be unique. In this sense, scope refers to the type of the identifier. For example, identifiers of constant declarations do not need to be considered when examining the uniqueness of type declaration identifiers. The following declarations must have unique identifiers within their own scope: class, type, constant, association, aggregation, and global subprogram. Additionally, events within each class’s dynamic model must be unique, unless the intent is that the non-unique events represent an event that is shared between the classes. The purpose of this rule is to ensure that there is no ambiguity between identifiers of the same type. For example, if classes were allowed to be named the same, destination of intra-model event communication would be vague as to the appropriate receive class. This rule becomes the basis for the effort to deconflict input model identifiers prior to integrating.

2.2.3. Analysis Type Declarations. It should be noted that since AWL is a wide-spectrum language, there are type declarations that reflect different stages in the software life cycle. However, the point at which the domain model is involved within the software system creation process is in the analysis phase. Because of this, there may be type declarations that are

not allowed within the scope of the integration methodology. Specifically, the access, record, and union types are not addressed, as they belong to the design phase. This leaves allowable types that fall into the following categories: abstract, enumeration, integer, real, and container. Each of these types have rules defined that govern specifically how the conversions between them are handled.

2.2.4. No Class Attributes. Classes cannot have attributes that reference other classes. While this is a common and accepted practice in the designing and coding of software systems, it cannot be allowed in the analysis phase of software development. If information from one class is required in another, that information must be specifically requested and received through parameters on send and receive events. In order to represent the parent-child relationship, classes are required to use the aggregate relationship structure. The benefits of using the aggregate structure include the ability to easily modify models. Without this standard, modifying classes could cause difficult to detect side-effect logic errors within the parent class(es) of the modified child class(es). In addition, by adhering to this rule, the application engineer is able to provide any necessary parent and child membership constraints by using either the aggregation structure or invariants within the model's system class.

2.2.5. Actions on Transitions in the Dynamic Model. Actions or procedure calls are only allowed in the dynamic model of each class, specifically on the transition from one state to another. In addition to tying the class's dynamic model to its functional model, this rule regulates when actions can be performed. Without this restriction, there would be ambiguity within the model as to when or how actions are performed, and therefore, control over the class's attributes would be weakened. For example, suppose procedure calls could be nested within other procedure's post-conditions. If so, the calling of any given procedure could have unexpected results through the performing of unintended actions.

Additionally, actions that are allowed to occur outside of the transition would make event communication meaningless in the integration process. It is important to see that other than the structural merging of the input models, the dynamic model integration is the purpose for using this methodology. It is the event communication processes that truly provide the functionality of each input model, and it is the combining of these processes in various ways that allow the merging of multiple models to solve bigger or more complex problems.

2.2.6. Functions are Not Actions. There is a clear distinction between functions and procedures. While they are both subprograms, they provide the model with different abilities. Procedures are the class's actions executed on transitions triggered by events within the class's dynamic model. Functions, on the other hand, provide the application engineer with the ability to simplify the expressions within constraints in invariants and pre- or post-conditions. By using functions, the engineer is able to insert complex expressions into any constraint's boolean expression. Function calls, unlike procedures, return a value which can be used in a boolean expression. As such, function calls are in fact expressions themselves. Functions can be defined locally or globally depending upon their intended scope. For example, a function that does not use any variables except those passed in through its parameters or non-globally defined constants can be defined globally. Like other subprograms, classes can only reference locally or globally defined functions.

2.2.7. Restricted Class Visibility. Information that needs to be shared between classes must be passed via event communication. This rule is enforced to ensure that each class is responsible for its own information. Attributes within a class may be kept up-to-date in any way that the designer so chooses. More specifically, accessing a class's attributes directly might not ensure that an expected value is retrieved. To avoid this problem, information (attribute values) is strictly communicated through the use of event parameters. Additionally, a class's defined

subprograms are likewise restricted to being called only within that class via the dynamic model. This also protects the integrity of each class's attributes as well as avoiding difficult to detect side-effect logic errors caused by modifying subprograms that are called by classes other than the originating class. Note, all classes have visibility to globally defined constants and subprograms.

2.2.8. Intra-Model Event Associations. This house-keeping rule was established to assist the application engineer in identifying intra-model event communications. Those events that have at least one send and receive class within the domain model in which they exist must have a corresponding association between the involved classes. Not only does this enable the engineer to quickly identify those events that are designed as inter-model communications (events without such associations), but it has other benefits as well. First, it allows for the definition of invariants over event associations through the association structure or in the model's system class. Secondly, it precludes ambiguity over the target of intra-model events as the receive class(es) is(are) defined in the association structure through the use of the structure's role identifiers. Note, AWL's association syntax does not require or offer specific role identifiers for associations used to denote intra-model events. Therefore it is left to the domain expert to use suitable role identifiers such as: "receiver", "r", "r1", "sender", "s", "s1", etc. to describe the role of the class in the relationship.

2.3. Software Component Theory

This section focuses on how domain models can be used. The assertion is that domain models behave in a similar fashion to software component theory. Specifically, domains can be looked at in the same light as software components as described by Szyperski [2]. The power of a reusable component is that pieces of systems or entire systems that are already created do not have to be reinvented [1, 2]. Additionally, libraries can be created that application engineers can

utilize in the creation of new systems in support of user generated requests. Components have a wide definition that includes many different disciplines. Component terminology typically refers to software bundles that software, or other, companies can exchange or sell that perform some specific task.

This technology can roughly be compared to electrical engineering. For example, suppose a group of electrical engineers were designing a bread-board to solve a given problem. They would not design and implement all of the lower level components. Rather, they would purchase a chip with all of the needed components on it. Their creative energies would then be spent on designing a new configuration of the basic building blocks provided by the chip. Similarly, a construction crew could save a lot of time constructing a building by using pre-assembled modules with the added benefit of being sure that the end result was desired by the customer.

When thinking about domain models, it is useful to consider models as a type of software component. Specifically, by viewing domain models as components, particularly with regard to the concepts of independent deployment and interface contracts, application engineers are released from the burden of becoming domain experts.

2.3.1. Component Properties. Independent deployment is synonymous with another term, black-box. Black-box in any discipline implies that the internals of the object cannot be examined or modified. In the context of domain models [1, 2, 3, 12], this means that models are not developed to interact with any other specific model. However, they can be designed with the knowledge that they will interact with other models, perhaps making integration easier, but in no case should they reference or know specifically of other models. In part, this software component rule is enforced indirectly because domain experts create models that exactly specify the construction and behavior of their respective systems. It is important to note that domain experts cannot foresee the many possibilities of how their model could be utilized. Because of this,

domain integration will have to deal with deficiencies between models to match the requirements of the new system. This issue is addressed by the interface contracts, and resolved in the methodology. Another reason for the black-box approach is to protect the correctness of the input models. Because the models were created by domain experts, the confidence in the validity of selected input models is high. If we were not confident that the models were correct, they would be of no use to us and the application engineer would have to search for a different model or worse, create one from scratch. The idea that models could be modified leads to serious consequences in terms of re-verifying the model's correctness.

2.3.2. Interface Contracts. In keeping with the building analogy, before attempting to integrate the selected components, the contractor must consider two things. The first is to determine that the building modules, when combined, satisfy the requirements of the intended building. The second is to determine that the modules have compatible interfaces [1, 2]. The consequences of using a module within a building that does not meet the established interface requirements could be devastating to the stability of the structure. In terms of software systems, messages that do not meet the published interface contracts could cause system crashes. Errors in matching messages fall into two categories, incorrect or missing events and incorrect or missing parameters. Critical to a successful integration is the process of first identifying that a domain model is appropriate for integration, accomplished by either the application engineer's first-hand model knowledge or more likely by examining the model's documentation such as its abstract, and secondly, by accepting the model's interface contract.

What is the contract? A contract in the sense of a domain model is centered around the model's events. Models will have some combination of inter- and intra-model events. The inter-model events are events that are sent or received by the domain model to or from another domain model. Additionally, each event, either inter- or intra-model, has a list of parameters associated with it. A model's interface contract is the formal definition of these events and their parameters.

The distinction between internal and external events is made to aid the application engineer in identifying starting points for the model integration, but the contract is a published definition of all events within the model. Figure 9 represents the interface contracts for Models 1 and 2, by formally describing the models' inter- an intra-model events along with their associated parameters including the necessary type definitions.

```
type myInt is range 0 .. 100;
type weekDays is (Mon, Tue, Wed, Thr, Fri);
type CHARACTER is abstract;
type STRING is sequence of CHARACTER;
```

```
var1 : myInt;
var2 : weekDays;
var3 : STRING;
```

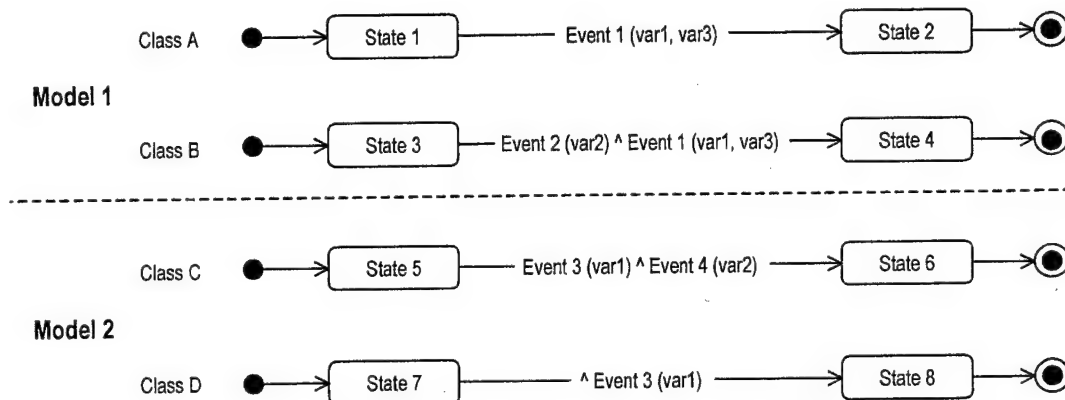


Figure 9. Domain Model Interface Contract Example

2.4. Type Theory

Another background topic of considerable concern to the successful integration of domain models is that of type theory. Two inter-related topics can be derived from this particular issue. The first challenge is to ensure that the target value of a converted variable is compatible with the range of the target variable type. The second challenge is to ensure that the conversion between differing types is both possible and correct. Both of these concerns need to be addressed in order to successfully create a correct conversion between the interface contracts of the involved

input models. In order to discuss the possibilities of type and value conversions, a review of range restrictions is required. The correct conversion of the parameters is possible only through strict adherence to the rules of range restriction.

2.4.1. Range Restrictions. The concept of range restrictions, as discussed in the context of type conversions, is based upon the comparison between the range of values of both the source and target variables. The range of the target variable is compared to the modified value of the source variable to determine if the conversion falls within the range restriction. All possible values of the source variable, when converted, must fall within the target variable's range in order for the type conversion to be valid. There are several distinct possibilities for the outcome of the range comparison between the source and target ranges. The categories are: disjoint, overlap, equal, weak-to-strong, and strong-to-weak. Ranges that are disjoint from each other have no common range values. Ranges that overlap, on the other hand, do share some common range values, but both the source and target ranges have some range values that are not included in the other. While disjointed ranges have nothing in common and overlapped ranges share some portion, the weak-to-strong comparison has the missing values in the target range only. The last two categories, equal and strong-to-weak, are similar in that every source range value is accounted for in the target range. Figure 10 depicts a Venn diagram illustrating each of these categories.

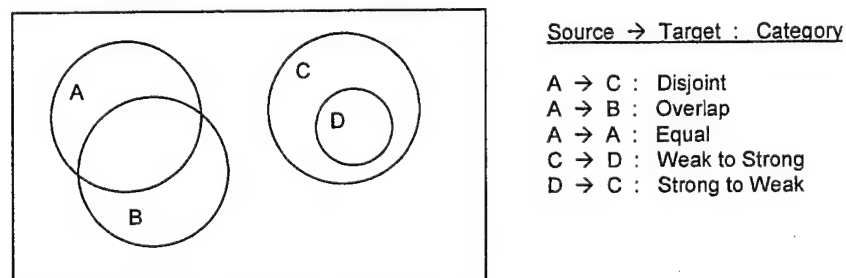


Figure 10. Source-to-Target Range Categories

The Venn diagram graphically illustrates the source and target ranges as sets, while Figure 11 adds to the demonstration by providing some concrete examples. The example, including Table 3, shows each of the categories, as well as possible ways of affecting the target value such that the result can be moved from one category to another.

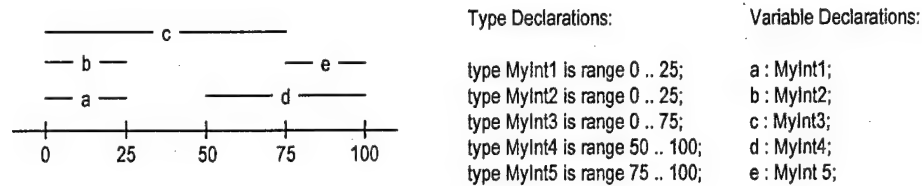


Figure 11. Range Restriction Examples

Table 1. Valid and Invalid Type Casting Examples (based on Figure 11)

Valid	Invalid
b = MyInt2'(a) – equal	d = MyInt4'(b) – disjoint
e = MyInt5'(a + 75) – equal	c = MyInt3'(d) – overlap
c = MyInt3'(a) – strong-to-weak	e = MyInt5'(c) – weak-to-strong
c = MyInt3'(e - 25) – strong-to-weak	b = MyInt4'((e - 75) * 2) – weak-to-strong
a = MyInt1'(d / 4) – equal	

As an additional example, consider the following scenario. Model A produces a parameter “area” of type Integer with a range of 0 to 1000, and model B requires a parameter “theArea” of type Real with a range of 0.0 to 1000.0. In this case, the conversion between “area” and “theArea” is straightforward, because the range comparison is strong-to-weak, meaning that for every value in the source variable there exists a corresponding value in the target. There is no harm in the fact that not every target range value can be reached. Additionally, it is important to recognize that the range comparison is accomplished after any value conversion or computation is applied. For example, minute and second types could not be directly compared. Rather, the variable for the minute would need to be multiplied by 60 in order to correctly compare to the second. Thus, if the range of the source (minute) were 0 to 100, then the range for the target (second) would have to 0 to 6000 in order to be equal.

2.4.2. *Type Bridging.* Range restriction is an important concept to explore with respect to type bridging because it provides the application engineer with a tool to determine how to go about bridging the gap between the supplying and the receiving models. If the range comparison between the source and the target are compatible, either equal or strong-to-weak, then the integrator has the option to use a type casting operation. However, if the ranges are disjointed, overlap, or are weak-to-strong, then a manual mapping must be applied such that all of the source values are accounted for. Table 2 defines the requirements for both the type mapping and type casting forms of type bridging.

Table 2. Type Mapping and Type Casting Definitions

Type Mapping	is a...	conjunction of expression implications.
	Expression Implication	a source expression, an implication, and a target assignment.
	Source Expression	an expression that selects a range of values from the source variable's type definition by using a comparison operator(s).
	Target Assignment	the target variable, the assignment operator, and a valid value from the target variable's type.
	Constraint	All elements within the range of the source variable's type definition must be represented once and only once in the mapping's expression implications.
	Example	source expression1 \Rightarrow target assignment1 \wedge source expression2 \Rightarrow target assignment2 \wedge source expression3 \Rightarrow target assignment3 \wedge ... source expressionN \Rightarrow target assignmentN
Type Cast	is an...	assignment statement.
	Assignment Statement	the target variable, the assignment operator, the target variable's type, the casting operator, and the source conversion expression.
	Source Conversion Expression	an expression that modifies the value of the source variable to be compliant with the target's type.
	Example	target = target type' (source conversion expression)

In addition to type range restrictions and the method of type bridging, there is another issue to deal with. There are rules that are dependent upon the types of the source and target variables. Table 5 provides a complete list of the type combinations as well as issues that must be considered in each case. It is significant to point out that in all cases, type mapping is allowed. However type casting is only allowed during Integer and/or Real type conversions, provided that the range restrictions agree. Additionally, classes, because they can be passed as parameters, are also considered in Table 5, but cannot be converted to any type. Remember that classes do not have visibility into other classes, therefore the conversion classes do not have the ability to discern any information from an incoming parameter-class, which would be required to perform a type mapping.

Table 3. Type Bridging Issues (Combination of Type Conversions Allowed)

Integer-to-Integer	<ul style="list-style-type: none"> ▪ There is no loss of precision. ▪ Type casting allowed if range restrictions are met.
Integer-to-Real	<ul style="list-style-type: none"> ▪ There is no loss of precision. ▪ The precision of the converted type's value is increased to the delta in the target type's definition. ▪ The upper and lower bounds of the target type are type cast to Integer in order to compare the source and target's ranges. ▪ Type casting allowed if range restrictions are met.
Real-to-Integer	<ul style="list-style-type: none"> ▪ There is a loss of precision. The precision of the source variable is reduced to an Integer value. ▪ Note, because of the loss of precision, a problem can occur if the modified value is in turn returned to the supplying model. ▪ The upper and lower bounds of the source type are type cast to Integer in order to compare the source and target's ranges. ▪ Type casting allowed if range restrictions are met.
Real-to-Real	<ul style="list-style-type: none"> ▪ No loss of precision. ▪ Type casting allowed if range restrictions are met.
Container-to-* or Enumerated-to-*	<ul style="list-style-type: none"> ▪ Type mapping is the only allowable bridging technique. ▪ Container type includes sets, bags, and sequences.
*-to-Container or *-to-Enumerated	<ul style="list-style-type: none"> ▪ Type mapping is the only allowable bridging technique. ▪ Container type includes sets, bags, and sequences.
Class-to-Class	<ul style="list-style-type: none"> ▪ While classes can be passed as parameters in the object-oriented paradigm, they can not be modified by the conversion process. Therefore, the bridging in this case is to simply pass the class from the source to the target

3. Domain Integration Analysis

Chapter 2 provided background to help understand the problem space (Sections 2.1 and 2.2). It also explored additional topics (Sections 2.3 and 2.4) necessary for the solution of the problem identified in Chapter 1. This chapter, Domain Integration Analysis, is the bridge that closes the gap between the problem analysis and the presentation of the solution in terms of the integration methodology presented in Chapter 4. In order to accomplish this goal, this chapter is divided into a progression of refined hypotheses starting with initial assumptions and concluding with the final solution of generating model interface conversions.

3.1. Initial Hypothesis

The starting hypothesis centered around three core ideas: domain model selection, hooks, and middleware. These concepts were used as the initial approach in finding a solution to this problem. Together they represented the totality of the solution. However as will be seen, they required further analysis before they could be expanded to provide the insight needed to develop an appropriate integration methodology.

3.1.1. Domain Selection. Probably the most important step in the domain model integration methodology is the process of selecting the models to use. There are several important aspects to selecting the right input models for the integration. The first step in the process is to understand the problem statement. In that way, input models can be selected with the purpose of satisfying specific problem statement requirements. The purpose of domain integration is to assist the application engineer in forming a formal specification that fulfills the requirements of a given problem statement. In other words, the domains must make sense to be considered for entry into the integration process. An additional concern is finding potential

models. The application engineer must have access to some store of domain models, and more importantly, the models' descriptions. Models must be available, easily understood (through published abstracts and interface contracts), and they must be well-formed. Only after meaningful, appropriate, and well-formed models have been selected can the integration process begin.

The engineer must have a good grasp of the problem statement so that a high-level solution can be derived. Additionally, the portions of the problem statement that are satisfied by utilizing an existing domain model must be identified. It is these identified portions of the problem statement that must be matched with primary purpose or secondary effects of candidate models. While this step is not addressed specifically in this research, it can be seen that there is considerable skill involved in dividing a complex problem statement into separate and distinct areas that have the possibility of reusing existing models.

The application engineer will explore available domain model libraries to select models that fulfill some or all of the problem statement's requirements as identified in the previous step. To this end, each model must have the following documentation available:

- What is the primary purpose(s) of the model?
- What are any secondary effects of the model?
- What communication process(es) supports the primary objective?
- What communication processes support the secondary effects?

3.1.2. Hooks. Using the software component theory as a starting point, the initial thrust of this research focused on the need to find "plugs", to use an analogy from an electrical appliance, or "hooks" into the selected domain models. This decision was made to ensure that each model was treated as a distinct entity, enabling users to combine multiple components together without having to know what was inside, only how to plug it in. The hook concept supposed that models must have some kind of recognizable external interaction beyond the

model. These external interactions become the hooks into each model allowing their inter connection to other models.

Identifying what the hooks were was straightforward. They had to be the model's events, as they provide the communication between the model's classes (see Section 2.1.3).

Unfortunately, determining which of the model's events should comprise the hooks is not readily apparent. The problem is that models may have many events, and determining which ones to use is a difficult task unless the application engineer has a detailed understanding of the model. It is precisely this detailed model knowledge that the methodology is supposed to address.

Additionally, once the hooks are found the application engineer must devise a way to use them. While it seemed fairly obvious that the events were the key to connecting domain models, the problems of identifying them and determining what to do with them still needed answering.

3.1.3. Middleware. As the previous section indicated, the goal of this approach was to hide the internal operation of the input models as in the black-box software component theory (see Section 2.3). Thus, the connection between the input models must be accomplished using only their events. In order to facilitate this connection, the conclusion was reached that some additional software component was required to handle the differences between the hooks. The events identified as hooks were chosen because they contained or required information pertinent to their model. Therefore the new software must contain some mechanism to convert the shared information from one form to another. The middleware concept was born from these ideas.

Ensuring the correctness of the input models was protected using this approach. However, now the middleware would also have to verified to be correct or created in such a way that correctness was guaranteed. The problem was determining how to correctly convert the information supplied by one model and required by another. Correctness preserving conversions over type bridging operations are explored in Section 2.4. In addition to providing the conversion between values sent from one model to values received by another model, the middleware would

have to be robust enough to handle a myriad of variations in the way events could be combined to facilitate model integration. The problem arises that the application engineer would be responsible for creating the middleware in an ad hoc fashion, meaning that he or she would require an in-depth knowledge of the input models. Clearly, the next step would be to identify the categories of integration in order to standardize the creation of the conversion middleware. Another concern about standardizing the middleware was determining what the functional and dynamic models should look like. The following questions represented some of the concerns that arose because of the middleware approach. What should be done with events received by the middleware? How are converted values generated? When can the middleware's send events be generated? The following section, Refined Hypothesis, addresses the issues created by the initial hypothesis.

3.2. Refined Hypothesis

After the initial approach was identified, the effort focused on answering the questions and challenges presented by the first set of hypotheses. While the domain selection stood on its own because of the assumptions and scope of this problem, the remaining concepts of hooks and middleware needed further development. The idea of hooks evolved into inter- and intra-model events and focused on the use of model interface contracts. Additionally, an analysis of communication patterns between models using events was conducted. This study provided insight into communication patterns, event connections patterns, and event parameter patterns. These formed the basis for defining a standard middleware object. The middleware evolved into a Model Interface Conversion (MIC) class that was responsible for handling the predetermined patterns. In order to answer the questions of how the conversions were generated and when such generations could occur, an additional study was conducted that focused on triggering these actions from the MIC's dynamic model.

3.2.1. Interface Contracts. As explained in Section 2.3.2, the components, in this case models, have and must adhere to contracts of behavior. One of the problems identified in the initial hypothesis was that the application engineer would be required to fully know the domain models in order to pull out the model's interface. There are two approaches to solving this problem. The first is to require the domain expert to publish a contract along with the model's documentation. This would serve to assist the application engineer in selecting appropriate models for integration. The second method would be to use an automated support tool to identify the interface contract. This seems wholly feasible as the information is contained in total within the model in specific locations. Both methods should be used simultaneously, allowing the application engineer to perform a high level analysis over possible input domain models while freeing him or her during the actual integration of the input models. The interface contracts must identify all events used by the model. Additionally, parameters that are associated with each event must also be scrutinized to ensure that the required input/output variables are dealt with.

Table 4. Example Model Interface Contract (Model 1 from Figure 9)

Event	Inter/Intra	Class	Description
	Parameters	Parameter Type	
Event 1	Intra		<i>Place any comments that help explain the purpose of the event and any of its parameters here.</i>
	var1	myInt	
	var3	STRING	
Event 2	Inter – Class B (receiver)		<i>Place any comments that help explain the purpose of the event and any of its parameters here.</i>
	var2	weekDays	

Table 5. Example Model Interface Contract (Model 2 from Figure 9)

Event	Inter/Intra	Class	Description
	Parameters	Parameter Type	
Event 3	Intra		<i>Place any comments that help explain the purpose of the event and any of its parameters here.</i>
	var1	myInt	
Event 4	Inter – Class C (sender)		<i>Place any comments that help explain the purpose of the event and any of its parameters here.</i>
	var2	weekDays	

In order to demonstrate how interface contracts are defined, Tables 4 and 5 are provided based on Figure 9. In this scenario, Model 1 has two classes: Class A and Class B, and Model 2 also has two classes: Class C and Class D. Additionally, each model has two events (representing both intra- and inter-model communication) complete with parameters. The demonstration illustrates how the interface contracts, published by each model, can be defined. In the example, Event 1 is an intra-model event between Class A and Class B. However, it is not important to keep track of the send and receive classes for intra-model events. Likewise, Event 2 is an inter-model event received by Class B, Event 3 is an intra-model event, and Event 4 is an inter-model event sent by Class C.

Thus, the interface contract for a given model outlines exactly what information the application engineer needs to know when connecting/integrating models. By looking at the inter-model events, he or she is able to determine what events, and therefore what parameters or information, any given model is guaranteed to produce. Likewise, the engineer can determine what information any given model requires in order to satisfy that model's inter-model receive events. It is this concept of interface contracts that is the basis of the creation of the MICs over the identified inter-model connecting communication patterns.

3.2.2. *Conversion Trigger Strategies.* The final topic to complete the analysis portion of the communication plan is to determine how to control when the MIC can generate the send event(s) from the gathered information. This is a critical piece of the MIC's dynamic model because it provides the guard condition that controls the transition between gathering the information and processing the information. There are four types of controls: counter-based, event-based, value-based, and user-based. Figure 12 is provided to illustrate the control flow that is established for the three strategies examined by this research. Each of these strategies dictate how the "trigger evaluation functions" are created. Trigger evaluation functions are discussed in the section concerning the MIC's functional component in Chapter 4. However, in order to understand why those functions are needed, this section provides analysis to assist the application engineer in selecting an appropriate trigger strategy for each of the communication patterns in the integration plan.

3.2.2.1. *Counter-Based Trigger Strategy.* The important information in the counter-based trigger strategy is how many events have been received by the MIC during the appropriate receive state in the dynamic model. This is a good technique for the merge and mixed communication patterns where the order of the receive events does not matter, and it is known how many events will be received. For example, suppose one model produces an event that supplies a dimension value and another model receives an event that requires an area dimension. In that case, it would be possible for the MIC to receive the same input event twice, calculate the area and generate the appropriate send event. Another example may be that the supplying model(s) produces two different events; one containing a length dimension and another containing the width dimension. If the events can be received in any order, then the MIC would have to wait for the two events before processing the area. The major drawback to this trigger strategy is that when there is uncertainty concerning how many or when different events will be

received by the MIC, it can make it difficult or impossible to correctly design the corresponding trigger evaluation functions.

3.2.2.2. Event-Based Trigger Strategy. Unlike the counter-based strategy, the event-based strategy does not keep track of the number of events received during the information gathering state. Rather, when a specified event is received, the trigger is set and the transition can be made which generates the send event(s). This strategy is perfect for the transfer and split communication patterns because only one event is received before the conversion takes place. However, it can also be useful if the MIC is to receive some unknown number of events, but upon receipt of a specific event it stops gathering and processes the outgoing information. This strategy also has its drawbacks. For example, if the intent is to trigger the conversion on the second or later occurrence of the receipt of an event, some other strategy is needed.

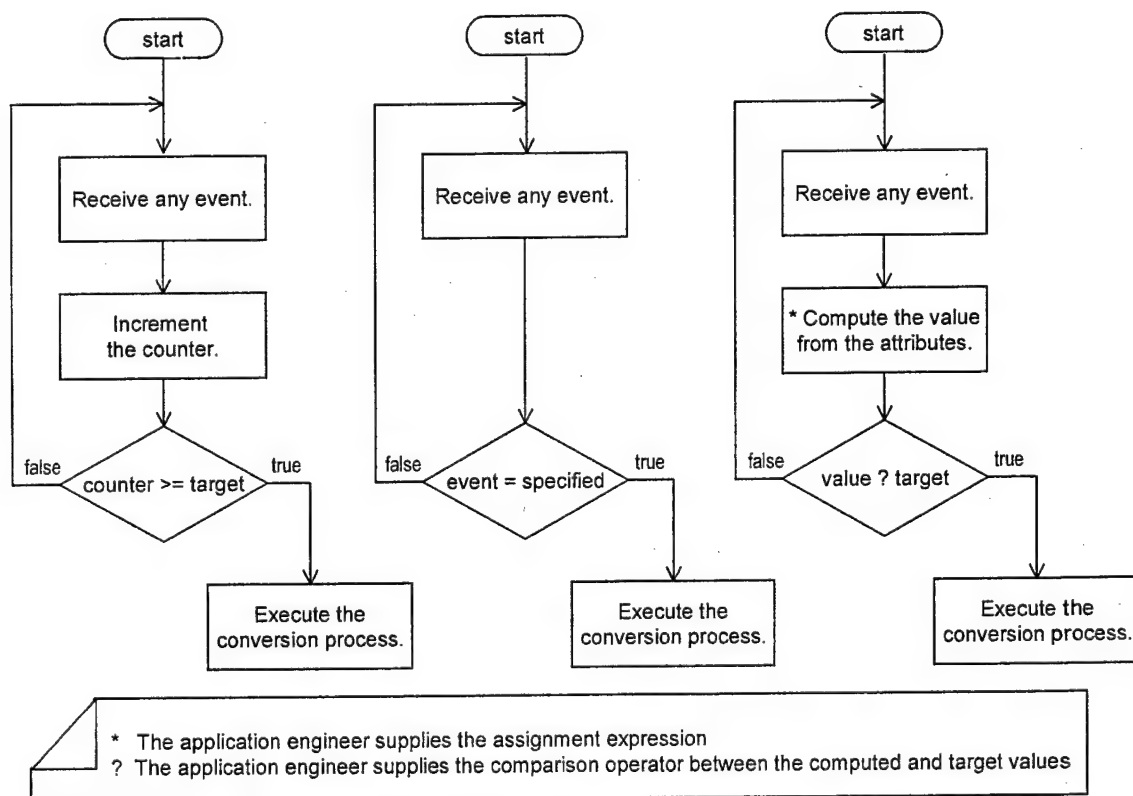


Figure 12. Counter-, Event-, and Value-Based Trigger Strategies

3.2.2.3. *Value-Based Trigger Strategy.* Yet another strategy is the value-based where the values contained in the MIC's attributes are used to determine if the trigger should be set. There is a lot of flexibility with this strategy in terms of what can be done with the computation of the MIC's attributes. However, this also means that the application engineer is required to provide a correct boolean comparison expression to check the value(s) against a specified target value. Another flexibility issue with this strategy is that the number of events received does not matter. Likewise, whether or not a specific event is received also does not matter.

3.2.2.4. *User-Based Trigger Strategy.* The last trigger strategy is the least restrictive. The user-based trigger strategy recognizes that any given communication pattern may have unique needs that are not completely satisfied by one of the previous three strategies. There are conceivably many different combinations of strategies that the application engineer could create by using the counter-, value-, and/or event-based trigger strategies. As such, the engineer is given the opportunity to create his or her own trigger evaluation function based on the previous themes. While creativity can be used to produce the necessary function, the engineer must limit the function to those three items due to the lack of any other available alternatives.

3.2.3. *Communication Patterns.* As part of the integration analysis, the application engineer must identify those communication events that are relevant to the integration of each model. This is done by studying each model's interface contract. These contracts provide a detailed description of what each event supplies or requires. Additionally, the published interface contract identifies the model's inter-model communications which are key to developing the integration plan.

Communication patterns refer to the overall view of how information is disseminated from one model to another. The patterns are a high-level view of how events need to be configured in order to achieve the functionality desired by integrating the input domain models. In the communication pattern, there is no concern about the type conversions or value computations required to facilitate the successful sharing of information between the models. Rather, each communication pattern focuses on identifying the point(s) in the supplying model(s) at which the desired information is available, and at what point the receiving model(s) requires the shared information.

Every input model in the integration must have some connecting communication with another model, such that all of the models are connected through the integration of their dynamic models. In other words, the goal of integration is that all of the domain models must be joined in one new model. It is the communication patterns between the input models that become the glue that binds the models together.

3.2.3.1. Transfer. The simplest of the communication patterns is the automatic *transfer* of a single event from one model to another event in another model. In this pattern, the application engineer has found an event that supplies all of the required information to completely satisfy a process in another model. One of the reasons that this pattern is simple, is because the trigger strategy of determining when to generate the send event can be based on either the receive event or an event counter with a target value of one.

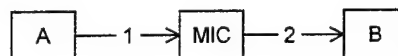


Figure 13. Transfer Communication Pattern

3.2.3.2. Merge. A slightly more complex communication pattern to identify and design is the *merge* pattern. In this communication pattern, multiple events are required to

supply the information necessary to generate the send event. It is similar to the transfer, but the trigger strategies for the merge can be more involved. Specifically, while the specific event or event counter approaches can be chosen, value-based trigger strategies also become available. There are two points to mention with regard to where the MIC's receive events come from. The first is that the multiple events are not restricted in terms of where they are generated. In other words, the MIC's receive events can be sent from one or more originating models. The second point of interest is that unlike the graphical representation in Figure 14, there is no restriction stating that the multiple events need to be unique. It would be perfectly legitimate for a MIC to receive many instances of the same event.

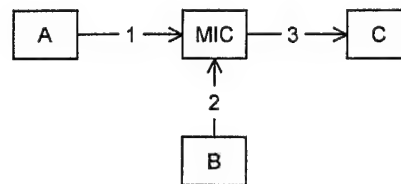


Figure 14. Merge Communication Pattern

3.2.3.3. Split. The *split* communication pattern is closer to the transfer pattern than the merge was, in that the MIC's send events are also produced automatically as soon as the receive event is processed. The difference is that, whereas the transfer generated one MIC send event, the split pattern generates multiple send events. Like the transfer communication pattern, the trigger strategies that are possible focus on the receipt of a specific event or an event counter with a value of one. Also, like the merge pattern, the MIC's send event's targets/destinations are not limited to one model. However, it is important to point out that the MIC only generates one set of events at one time. This is significant because after the events are generated and sent, the local attributes that were used to create the send event parameters are re-initialized. This initialization is done to allow the MIC to be ready for the next round of communications. However, as is shown later, the MIC could generate different send events at different times. This

is generally more complicated, and could indicate that the MIC may not be minimal enough. If this is the case, then unique initialization must occur to avoid losing necessary information for generating future send events.

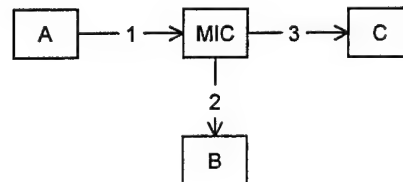


Figure 15. Split Communication Pattern

3.2.3.4. Mixed. If the transfer communication pattern is a special case of both the merge and the split, they are in turn both special cases of the most generic communication pattern, the mixed. The mixed pattern combines the more complex possibilities of conversion trigger strategies of the merge and the multiple generated send events of the split. However, like the split, it only generates send events once before the MIC is completely reset. The possibility that a MIC can generate send events at multiple points (non-minimal MIC) in its dynamic model leads to the last type of communication pattern.

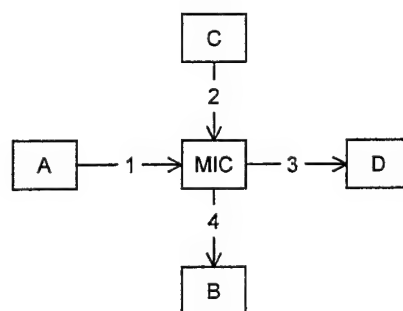


Figure 16. Mixed Communication Pattern

3.2.3.5. Combined. The combined communication pattern reflects the need to expand the possible formats in which MICs can operate. The first four, transfer, split, merge, and mixed, all reflect strictly minimal communication patterns where the flow of information within

the MIC's dynamic model was in one direction. First information is received through receive events, then it is processed (after a conversion trigger is set) into event parameters, and finally sent from the MIC. However, there may be cases in which the MIC must deviate from the standard flow. In these cases, the MIC may be responsible for prompting one or more models to provide additional information in order to complete the conversion of shared information. Thus, the *combined* communication pattern allows the MIC to generate interim send event(s) that gather additional information or signal some other action. Note that each cycle consisting of receiving events, setting a trigger condition, and generating send events, can be modeled as separate MICs except when the information gathered from previous cycles is required in the generation of send events in later cycles. The example in Figure 17 can be seen in this light. The receive event 1 from model A triggers the MIC to generate the interim send event 2 to model B. Later, the MIC receives event 3 which is then processed along with information gathered from event 1, and the final send event 4 is generated and sent to model C.

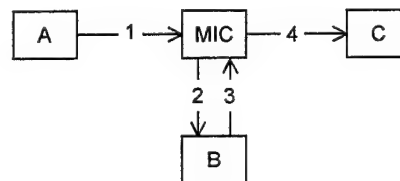


Figure 17. Combined Communication Pattern

3.2.4. Event Connection Patterns. An important integration concept is the idea of inter-model and intra-model communications. Simply put, send and receive events are used within models to get work done. Those communications within models that are relevant to that model have both the sender and the receiver within the model. These are the intra-model event communications, and during the creation of the domain model are not thought of as interfaces beyond the scope of the model. On the other hand, most models exist to provide some service or

information. Thus, another type of event communications exists. This is the inter-model communication, and is found by looking for events that do not have both a sender and a receiver defined within the model. These are typically the basis or starting point when looking for ways to integrate models, although it is possible to integrate models based solely on intra-model events. The inter-model events are documented through the model's interface contract, and should be easy to find and understand how to use.

Now that the communication patterns have been identified and analyzed, each of the MIC's receive and send events needs its own analysis. Specifically, the application engineer needs to understand how to "hook up" the MIC's events to and from the selected models. Events that connect via inter-model events will need new event associations for the integrated model. Likewise, events that connect via intra-model events require that the existing event association be modified to reflect the new connection(s).

The following choices represent the ways in which events are connected between the supplying model and the MIC, and between the MIC and the consuming model.

3.2.4.1. Inter-Model Event Connections. In this case, either the MIC's receive event connects to a model's inter-model send event, or the MIC's send event connects to a model's inter-model receive event. Shown in Figure 18, the events are sent from or received by one class. However, the MIC's send event could just as easily be connected with multiple models' receive events. Using an inter-model event as a connector with a MIC causes that event to become an intra-model event in the newly integrated model.



Figure 18. Inter-Model Event Connections

3.2.4.2. *Intra-Model Event Connections.* As described earlier, the domain experts cannot know or imagine all of the possible uses for their models. Therefore, there may or may not be appropriate inter-model events in the model's interface contract. Also, if an event that is intended to be a model interface is also required elsewhere in the model, it becomes an intra-model event. In any case, the connecting communication patterns can use these events as well. The only difference is that an existing event association must be modified to account for the new send or receive MIC. It is wholly possible for a MIC's send event to be connected to one model's inter-model event and to another model's intra-model event.

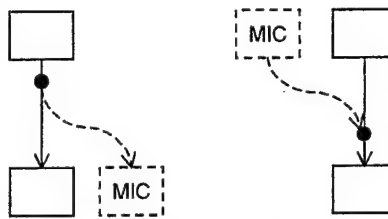


Figure 19. Intra-Model Event Connections

3.2.4.3. *Severed Intra-Model Event Connection.* The only allowable change to the input models is the severing of the model's intra-model communication. In this event connection scenario, the severed communication allows the insertion of another model's processing of the information into the existing communication path. Severing an intra-model event can be done for many reasons. One such reason might be to validate some information in the affected event. Another might be to hold or delay the processing of the affected event until some other event occurs. However, for whatever reason(s) this event connection is desired, it is important to maintain the validity of the affected model. It is not possible to interrupt or sever an intra-model event without completing the communication at some point. To do so would alter the input model in a way that could invalidate the correctness of the model.

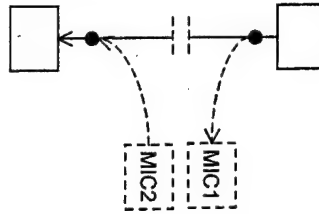


Figure 20. Severed Intra-Model Event Connection

3.2.5. Event Parameter Patterns. Another piece in the integration plan is to analyze the parameters in both the MIC's send and receive events. At the outset of this research there was a lot of studying concerning the relationship between the number of input to output parameters. The identified categories were: one-to-one, one-to-many, many-to-one, and many-to-many. While at some level it seemed significant that specific patterns can be found, the fact that all incoming event parameters are stored in the MIC's local attributes makes this topic purely academic. However, as just stated, there is no difference in terms of generating the MIC's send events, along with the event's parameters, as long as all of the required values were previously stored in the local attributes.

There is an issue of how to store the incoming parameters, be they one or a hundred. In order to preserve the integrity of the incoming data, and because in the general case it is unknown what processing is required to generate the output, each value must be stored individually. The first requirement is to store each incoming parameter in its own attribute. The second requirement is to store each parameter uniquely without affecting any previously received data. This is necessary because it is impossible to determine if the same event will be received multiple times. In order to satisfy both of these requirements, the MIC's local parameter attributes must be created as containers so that when the conversion occurs every received value can be accessed.

3.3. Final Solution

The previous two sections laid the foundations for the creation of the methodology. However, the final step of describing the results of the research still remains. The following sections outline the actions to be accomplished and structures required for the integration methodology. In order to create a new integrated model, three actions must be performed. The first action is the identifier deconfliction between the input models. Second, the new integrated model must be created. And the third, the required MICs must be created. Even after the merging of the input models under the aggregate structure of a single model, the integration is not complete. The real integration occurs after the MICs are created to add the functionality to the integrated model.

3.3.1. Domain Model Deconfliction. Perhaps the best place to start the domain integration is with the structural aspect of the integrated model. As the most basic part of the methodology, it provides the foundation for the new domain model. During this phase, the details of input model deconfliction as well as the creation of the new integrated model are worked out. Unlike the creation of the MICs, only the structural component of the input models can be modified during the creation of the new integrated model. The input models' functional and dynamic components are off-limits due to the previously discussed black-box software component theory. Additionally, the only modifications allowed are changes that do not affect the function or validity of the input models. Model deconfliction is an issue because the resulting integrated model must adhere to the same well-formed domain rules as the input models. While the input models are assumed to have passed the criteria of having all unique identifiers, it is the responsibility of the application engineer to ensure the validity of the output model. Therefore, the input models must be compared to each other prior to integration to address the concerns of ambiguity within the new model. For example, it would be perfectly legitimate for all of the

input models to share a common class identifier, call it *ClassA*. Also, assume that *ClassA* participates in communication in all, or some, of the input models. Under this situation, if the input models are combined without resolving the non-unique class identifier, there would be no way of knowing which instance of *ClassA* was participating in any given communication. Likewise, the same type of argument can be made for other "package-level" declarations such as type and constant declarations, global functions, associations and aggregations, as well as event communications.

There is a special case of deconfliction that should also be addressed here. Type declarations have an identifier, the deconfliction of which was just discussed, and a definition. The definitions, when considered across models, can fall into one of three categories: disjoint/overlapped, equal, or proper subsets. The cases where the type definitions are disjoint or overlapped are not significant when considering their deconfliction. Namely, they are considered to be separate and distinct types. However, the other two types, equal and proper subsets, offer the application engineer a possible savings in terms of eliminating a type declaration. If two type declarations between models have the same definition, then it would be correct for the integrated model to include only one of the declarations. For example, if multiple models have the type declaration: "type NATURAL is 0 .. *;," then the output model need only have the declaration once. However, there is a problem with this approach. If the type declarations matched completely, i.e. both the identifier and the definition match, then one of the declarations can be deleted. However, if only the definitions match, then by deleting one of the declarations the logic of one of the models would lose some of its meaning. Take the following as an example: model 1 has a declaration "type AGE is 0 .. 110;" and model 2 has a declaration "type SPEED is 0 .. 110;". If these two models were integrated and the declaration SPEED were deleted because of the matched definition with AGE, then wherever SPEED originally appeared in the model AGE would now appear. The result of the modification, while technically correct, would reduce the readability of the resulting model. Another possibility would be to combine the type

identifiers to produce "type SPEEDAGE". Again, the readability of the resulting model would be reduced. Additionally, while the readability of models may not be a concern in an automated system, in such a system, one extra type definition would also not be a concern. There is an additional concern that prohibits the use of this simplification. Type definitions that are proper subsets cannot be eliminated because the resulting definition would not be true to the original model design of the domain expert. The relaxed range restriction of the eliminated type could cause difficult to detect logic errors by allowing values that were previously forbidden with the original type definition. Therefore, only declarations that completely match can be simplified in the integrated model by only carrying one of the declarations forward. The same is true for the declarations of constants.

3.3.2. Integrated Model Creation. Creating the new integrated model has three steps. Step one is the creation of the new model, to include the creation of the integrated model's system class. The application engineer must supply the system class with a meaningful name as it will become the new model's handle. The next step is to copy each of the input models into the new model. The copy includes all of the classes, global subprograms, type and constant declarations, and associations and aggregations. The final step in the model creation is to tie the now copied input models under the umbrella of the integrated models' system class. This is accomplished by creating aggregate relationships between each of the input model's system classes and the new model's system class. The only information that requires the application engineer's intervention in this process is the name selection for the system class and the multiplicity of the children in the new system class aggregations. Along the same lines, the integrator can also create invariants over the new aggregates within the system class if desired. Once the new model has been created, or at least its shell, the input models can be removed from this process as they will no longer be needed.

3.3.3. *Model Interface Conversion (MIC) Analysis.* As shown in Figure 21, the identified communication patterns are shown as the “clouds” (there are two options presented in the figure) that encompass the events involved in the sharing of some information. Each of the clouds or communication patterns will signify the creation of a Model Interface Conversion (MIC). Option 1 indicates the creation of one MIC, while option 2 requires two MICs. MICs are the “middleware” that provide the necessary conversion between the supplying and receiving events. Because the input models are forbidden to be modified (and thus keep their integrity), the MICs become the agents of change that make the required connections possible. The communication can be designed as one all-encompassing MIC or as a set of minimal MICs. The all-encompassing MIC groups all of the inter-connecting events into one large conversion process. The minimal approach, on the other hand, proposes a conversion process for each of the communication patterns that support the sharing of a common unit of information. What is meant by unit of information is the smallest set of events that are required to facilitate the sharing of one idea. For example, models that were chosen because they satisfy two or more aspects of the problem statement should have at least two MICs, one for each of the selection purposes. However, some communication patterns may require the composition of multiple smaller patterns, each sharing one idea, in order to create the information needed to share the one “master” idea. In these cases, an all-encompassing MIC is required. The decision whether to take the all-encompassing or minimal approach will directly affect the complexity of the required MICs. Thus, in order to create an easily understood integrated domain model, communication patterns should be identified such that minimal MICs are generated whenever possible. As is discussed in more detail in the following sections, the application engineer must make decisions on how to combine the provided information to produce the desired results. The goal of this methodology is to simplify, wherever possible, the responsibilities of the person(s) performing the integration. As such, this methodology assumes the use of the minimal MIC approach.

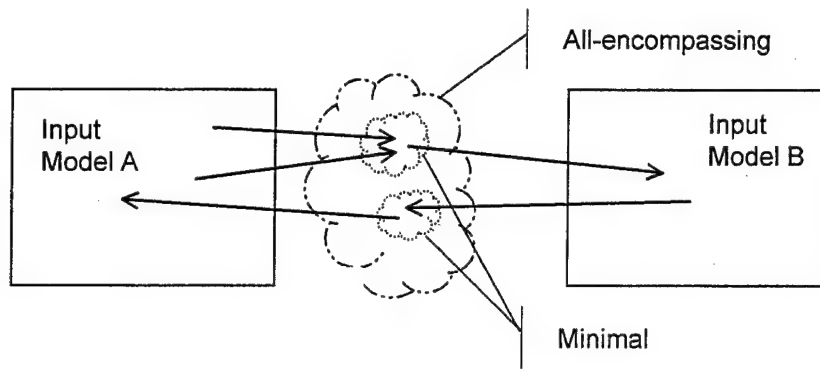


Figure 21. Identifying Communication Patterns (All-encompassing vs. Minimal)

The remaining sections of this chapter provide an analysis of the topics needed to successfully create a MIC, while Chapter 4 provides detailed descriptions of each of the MIC's components. MICs are created as classes in the integrated model, and as such have structural, functional, and dynamic components. As part of the structural model, MICs require local attributes to contain the parameters of the incoming events as well as other attributes that control the execution of the conversion. The functional model is represented by the need to have a conversion method and supporting type conversion and trigger evaluation functions. Finally, the dynamic model is represented by the need to control the sending and receiving of events.

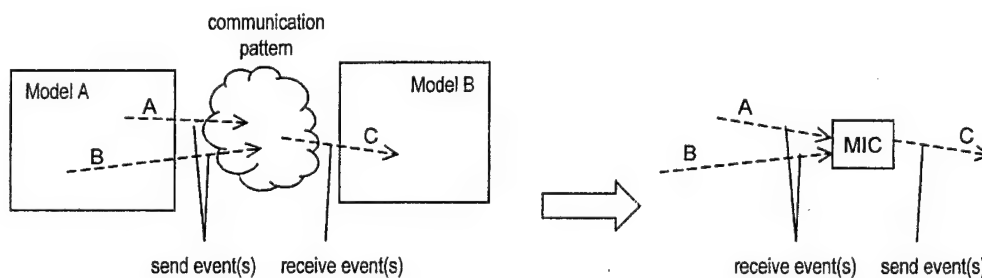


Figure 22. Input Model's Send/Receive Events become the MIC's Receive/Send Events

After identifying the communication patterns that will be developed into the new model's MICs, the engineer must then focus on analyzing each connection to determine the type of communication that is occurring. The following sections outline the possible communication

configurations. It should be noted that while discussing the MICs, the send and receive events are in relation to the MIC itself. Figure 22 illustrates how the integrated model's receive event(s) become the MIC's send event(s), and likewise the model's send event(s) are considered to be receive event(s) to the connecting MICs.

3.3.4. MIC Creation. Once the integration plan has been developed, and the input models have been copied into a new integrated model, the MICs can be created to complete the integration by implementing the connecting communication patterns. These classes, as previously mentioned, are composed of structural, functional, and dynamic components. Within the scope of the structural model, each MIC will have two types of attributes: parameter storage and trigger evaluation attributes. The functional component is comprised of the receive event processors including the corresponding trigger evaluating functions, and the conversion operation with its attribute-parameter converting functions. Finally, the dynamic model with its defined events, states, and transitions tie the MIC's structural and functional components together, and implements the connecting communication patterns that bind the input models together.

3.3.4.1. MIC Structural Components. Every MIC is composed of the same two types of attributes: those used to store the information in the receive event's parameters, and those used to assist in determining when to generate the MIC's send event(s).

3.3.4.1.1. Receive Event Parameter Attributes. Because the generation of the MIC's send events can occur at any time depending upon the analysis of the application engineer, the MIC must store all incoming information. Only after all of the necessary information has been gathered can the generation of the send event occur. Therefore, some method of storing the information in the parameters of the receive events must be decided upon. The first possibility is to create attributes of the same type as the incoming parameters. However,

this approach fails to capture the possibility of receiving multiple instances of the same event. In this case, the MIC could only over-write or add-to the existing value within the corresponding parameter's attributes. In order to preserve the integrity of all of the incoming information, the MIC must keep each incoming value distinct and reachable. Therefore, the receive event parameter attributes must be a container type, specifically a bag, to hold all incoming values. The bag allows the attribute to hold multiple copies of the same value to be processed later. It then becomes necessary to modify the structure of the new model by adding a new type declaration for each distinct parameter. The type will be a container of the parameter type. Once the new declarations are created, the MIC can create local attributes of the required type.

3.3.4.1.2. Trigger Evaluation Attributes. Along with the receive event parameter attributes, each MIC requires additional attributes to process the trigger strategy. For each point within the MIC that send events are generated, a new trigger attribute must be created. The trigger is a boolean type that is initialized to false and is evaluated to true when pre-described conditions are met. In addition to the required trigger attribute(s), the MIC may have additional attributes depending upon the trigger strategy chosen for the conversion. Conversions that are triggered by specific events would not require an additional attribute. However, conversions that occur after the receipt of a specified number of receive events would need a "COUNTER" attribute of type NATURAL. In the case where an additional attribute is required to process the trigger condition, the application engineer must ensure that the appropriate type declaration is present in the model, or create an appropriate one, and create the needed MIC attribute of that type. It should be noted that if the trigger strategy involves the comparison of local attributes to a specified value, a new constant declaration could be created within the model to store the specified value.

3.3.4.2. *MIC Functional Components.* Like the structural component, the functional aspect of the MIC also has an established set of items. Namely, all MICs have at least one "*ProcessReceiveEvent*" and "*Conversion*" procedure and one "*ConvertValue*" and "*EvaluateTrigger*" function. The following sections discuss the responsibilities of each of these subprograms.

3.3.4.2.1. *Conversion Initialization Procedures.* The first aspect of the MIC's functional component is its initialization. The initialization's purpose is to ensure that all of the local attributes are empty, and that any trigger attributes are reset, including setting the trigger(s) to false. This action, when executed, in effect resets the MIC back to its beginning point waiting for its first receive event. There is another type of initialization function that MICs can have. If the MIC has multiple trigger strategies, then there must be a unique initialization for those attributes that are involved in the generation of those send event parameters. All, some, or none of the local attributes can be reset during these initialization procedures depending upon the design of the application engineer in determining what is required for the processes final send events. In any case, after the last send events are generated, the "main" initialization procedure must reset the MIC to begin the cycle anew.

3.3.4.2.2. *Process Receive Event Procedures.* Every time a MIC receives an incoming event, certain activities must be accomplished. It is the responsibility of the "*ProcessReceiveEvent*" procedure to handle the tasks of storing the incoming event parameters, incrementing event counters, if necessary, and calling the trigger evaluation function. The first task mentioned is the main reason for this operation. In order to carry out this goal, the post-condition of the method must insert each of the receive event's parameters into the appropriate local attribute. In addition to its main task, the *ProcessReceiveEvent* method must also increment the appropriate event counter if the trigger strategy was to count receive events. The last function

of this method is to evaluate the trigger condition by calling the corresponding trigger strategy's "*EvaluateTriggerFunction*". In this fashion, the transition guard condition can be ready for evaluation at the beginning of any future transitions.

3.3.4.2.3. *Trigger Evaluation Functions.* The

EvaluateTriggerFunction returns the boolean value describing the state of the trigger condition. If the guard condition has been satisfied, then the function returns true; if not, the function returns false. As pointed out earlier, there are several types of possible trigger strategies, which directly affect the construction of the MIC's trigger evaluation function(s). The simplest function to create is the event-based strategy, where the function simply returns true because it is only called when the correct event is received. The other strategies require the application engineer to provide more detail in the creation of the post-condition's constraint expression. If the strategy was to generate the send event(s) after a certain number of events were received, then the constraint expression must have a comparison of the MIC's event counter to the newly created constant describing the number of events to receive. Still another possibility is the use of a value-based decision concerning the evaluation of the guard criteria. If this is the case, the engineer must again supply the required comparison expression between the local attribute's value to the constant containing the target value. This is more challenging because the local attributes can be computed in any number of ways to generate the value with which to compare the specified target value.

3.3.4.2.4. *Process Send Event Procedures.* At the point in the MIC's

dynamic model where the send events are ready to be generated, the "*Conversion*" method will be executed. The sole purpose of the *Conversion* method is to create the required parameters for all of the transition's send events. While it is the responsibility of this method to gather the appropriate output parameters, it does not generate them. Rather, each output parameter is

assigned a value by the execution of a corresponding *ConvertParameterFunction*. In this light, the creation of this method is easily automated because of its standard construction and the lack of application engineer input responsibilities.

3.3.4.2.5. Parameter (Send Event) Creation Functions. Unlike the *Conversion* method, the *ConvertParameterFunctions* rely heavily on the application engineer's input and creativity. These functions must return the required value after any and all of the needed type conversion and value processing has taken place. This is the place within the MIC that the actual conversions between the receive and send event parameters take place. The application engineer must use the stored values in the MIC's local attributes to generate the assignment expression that will return the expected result to the *Conversion* method.

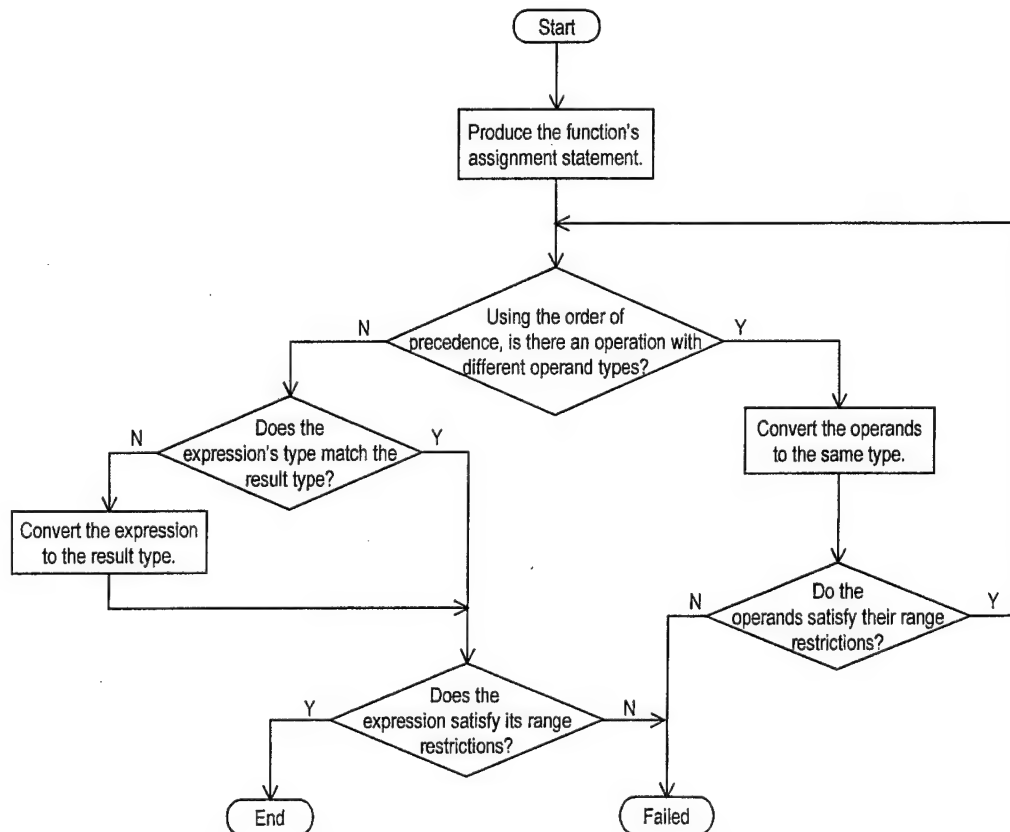


Figure 23. Convert Function's Assignment Expression Creation Flowchart

The creation of the convert function's assignment statement is of great importance. If the connection between the input models is flawed by an improper conversion the resulting model will be invalid, and the problem may be very difficult to identify. Therefore, special steps need to be applied to ensure that the supplied conversion logic is correct. There are two phases in the creation of the conversion assignment statement. The first is to generate the function's value computations, such as adding or in some other way combining the MIC's local attributes to generate the desired output. The second phase is applied to ensure that the resulting computations are type compatible. Previously, the topic of type bridging and type mapping were discussed as part of the type theory section. The integrator must correctly apply the appropriate type bridging technique to ensure the type compatibility goal is achieved. Figure 23 can assist the application engineer in successfully creating these complex assignment expressions.

3.3.4.3. *MIC Dynamic Components.* Once the MIC's attributes and the appropriate procedures have been created, there is only one step left in the MIC's development: the generation of the dynamic model. The dynamic model ties the MIC's functional components into a series of actions that, when performed in a described sequence, produce the desired transition between the selected input models. Using Figure 24 as an example, it is clear that there is an established pattern that can be applied as a template in creating all of the MIC's dynamic models. In the simple cases, where there is only one generation of send events, there will only be one iteration through the *ReceiveEvents* to *ConversionDone* states. Likewise, the state diagram will grow depending on the number of different events received by the MIC in each instance of the *ReceiveEvents* state.

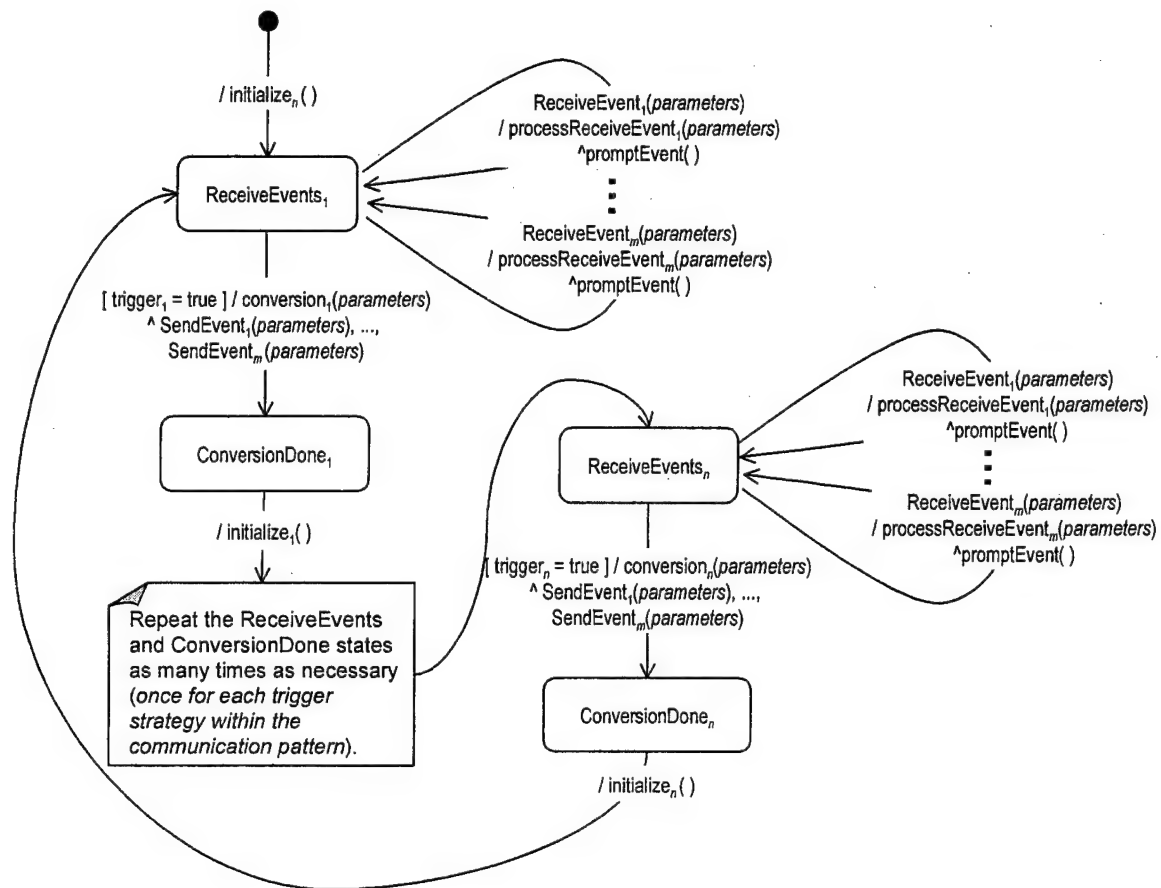


Figure 24. MIC's Dynamic Model State Diagram (Generic)

As outlined earlier, the MIC receives input events, which supply the MIC with the information necessary to generate the desired output events. The receive events are handled in the *ReceiveEvents* state, and are responsible for performing the *ProcessReceiveEvent* action. The events that can be received in any given *ReceiveEvent* state are controlled by the application engineer in the definition of the state diagram. By creating unique receive states, the designer is able to dictate the importance of only receiving particular events at specific times within the MIC's dynamic model. The send events are generated as the result of the *Conversion* action after a trigger condition has been met. There is one exception to the send event rule. If the flow of the dynamic model requires the MIC to prompt one or more models, the receive event transition can send a simple "prompting" event. If used, this event cannot have any parameters because there is no conversion procedure to generate them. While events are of utmost importance to the model

integration process, there is nothing additional to add as the previous section on the MIC's functional component describes them in more detail.

There are two states that exist in all MIC dynamic models: *ReceiveEvents_n* and *ConversionDone_n*. Each state allows only certain actions to be performed. The MIC can only receive incoming events in the *ReceiveEvents* state, which in turn causes the action to process the event, store the incoming parameters, and evaluate the trigger guard. When the appropriate trigger has been set, the MIC can generate and send the desired events. The transitions into and out of the *ConversionDone_n* state are automatic, which illustrates that the order of actions performed after the guard condition has been satisfied is always the same. The transition into *ConversionDone_n* performs the heart of the dynamic model's purpose, while the transition out of *ConversionDone_n* resets those attributes necessary to that particular conversion.

The transition table is the ultimate description of the dynamic model. It describes exactly what actions are performed on each transition and why it occurs. It is interesting to note that all transitions, except for the receive event transitions, are automatic. Once a trigger is set, there is a predetermined constant set of actions that always occurs.

4. Domain Integration Methodology

The preceding chapter, Domain Integration Analysis, provided a road map to understanding how domain models can be integrated. This chapter is dedicated to outlining a methodology that successfully integrates well-formed domain models. As Chapter 3 indicates, there is a lot of upfront planning that is required in order to create the required MICs. This fact leads to a division in the integration methodology that recognizes the information gathering that must occur prior to the creation of the integrated model. Section 4.1 defines the first three pre-integration steps that guide the application engineer through the analysis phase, while Section 4.2 describes the five-step integration methodology. Following the generic or UML based methodology, a language specific AWL version is provided in Section 4.3. Each step in the AWL methodology is traceable back to the UML version. Finally, the resulting methodology is then used to develop an automated support tool to assist application engineers performing an AWL domain model integration.

4.1. Integration Analysis

Before performing the integration of the input domain models there are some pre-integration steps that must be taken. While they are not actual steps of the integration, they do take into account the analysis that the application engineer must perform in order to successfully navigate through the methodology. The integration planning is conducted in three phases consisting of: input domain models selection, communication pattern identification, and communication pattern analysis.

4.1.1. Input Model Selection (Pre-Integration Step p1). Prior to any integration, the application engineer must select the input domain models. The models are selected based on the

requirements of a given problem statement. The models must be both applicable to the problem statement and have suitable interface contracts. Additionally, all input models must be verified to follow the well formed domain rules. During this process, selection decisions should be documented to assist in the next step, communication pattern identification. Figure 25 is a sample worksheet that captures the selection decisions for documentation purposes.

Domain Model Integration Worksheet Phase 1 (Integration Overview)
Project Identification: Problem Statement:
Input Model #1: Model description: (<i>The model is well-formed: Y / N</i>) Problem statement requirement satisfied:
Input Model #2: Model description: (<i>The model is well-formed: Y / N</i>) Problem statement requirement satisfied:
Input Model #3: Model description: (<i>The model is well-formed: Y / N</i>) Problem statement requirement satisfied:
Input Model #4: Model description: (<i>The model is well-formed: Y / N</i>) Problem statement requirement satisfied:

Figure 25. Sample Integration Analysis Worksheet (Input Model Selection)

4.1.2. *Communication Pattern Identification (Pre-Integration Step p2).* This step identifies the connecting communication patterns between the input models. The input models' interface contracts are used as the starting point for each of the communication patterns. The identified patterns should satisfy some portion of the requirements in the project's problem statement. Identification of the communication patterns is a very important analysis step because the decisions will be used later in the creation of the new model's MICs. The following sample worksheet is provided as an example of the types of questions that must be answered in order to correctly identify the communications that connect the input models.

Domain Model Integration Worksheet Phase 2 (Communication Pattern Identification)		
Interface Identification:		
What are the involved input models?:		
What is the information shared or purpose of this interface?		
What is the communication pattern (<i>circle one</i>)?		
<i>transfer</i>	<i>split</i>	<i>merge</i>
<i>mixed</i>	<i>combined</i>	
Sketch the communication pattern for the integration interface...		
Supplying		Receiving
Supporting		
Event → Model ○		
Model	Event	Inter, Intra, or Severed
Event	Parameter	Parameter Type

Figure 26. Sample Integration Analysis Worksheet (Communication Pattern Identification)

4.1.3. *Communication Pattern Analysis (Pre-Integration Step p3).* Once the application engineer has identified the communication patterns, he or she must analyze the necessary value conversions and trigger strategies. Again, a sample worksheet is provided that shows the questions that must be answered in order to successfully create the necessary MICs.

Domain Model Integration Worksheet Phase 3 (Communication Pattern Analysis)		
For each event the MIC receives...		
New Model Types (create a new container type for each unique parameter type):		
New MIC Attributes (create a local container attribute for each parameter in the receive event):		
Each group of send events the MIC generates must have its own trigger strategy		
Trigger Strategy	Attributes	Evaluate Function
counter-based	trigger count	$(func = true \wedge count = target) \vee$ $(func = false \wedge count < target)$
event-based	trigger	$func = true$
value-based	trigger	$(func = true \wedge targetValue ? userExp) \vee$ $(func = false \wedge targetValue complimentary ? userExp)$
other	trigger	supplied by the application engineer
For each parameter in the MIC's send event(s), create a Convert Function.		
Parameter	What is the logic (post-condition) that produces the desired output parameter from the available local attributes? (Consider both the type conversion as well as any value processing that needs to occur.)	

Figure 27. Sample Integration Analysis Worksheet (Communication Pattern Analysis)

4.2. Generic (UML) Domain Integration Methodology

Using the analysis conducted in Chapter 3, the following methodology was created. The methodology is a step-by-step process that takes two or more UML input domain models and creates a new integrated model. The integrated model is the result of merging the original models and creating the appropriate MICs and relationships to generate the functionality required to satisfy the presented problem statement. Figure 28 provides a visual representation of both the pre-integration steps as outlined in Section 4.1, and the remaining integration steps as described in the following sections.

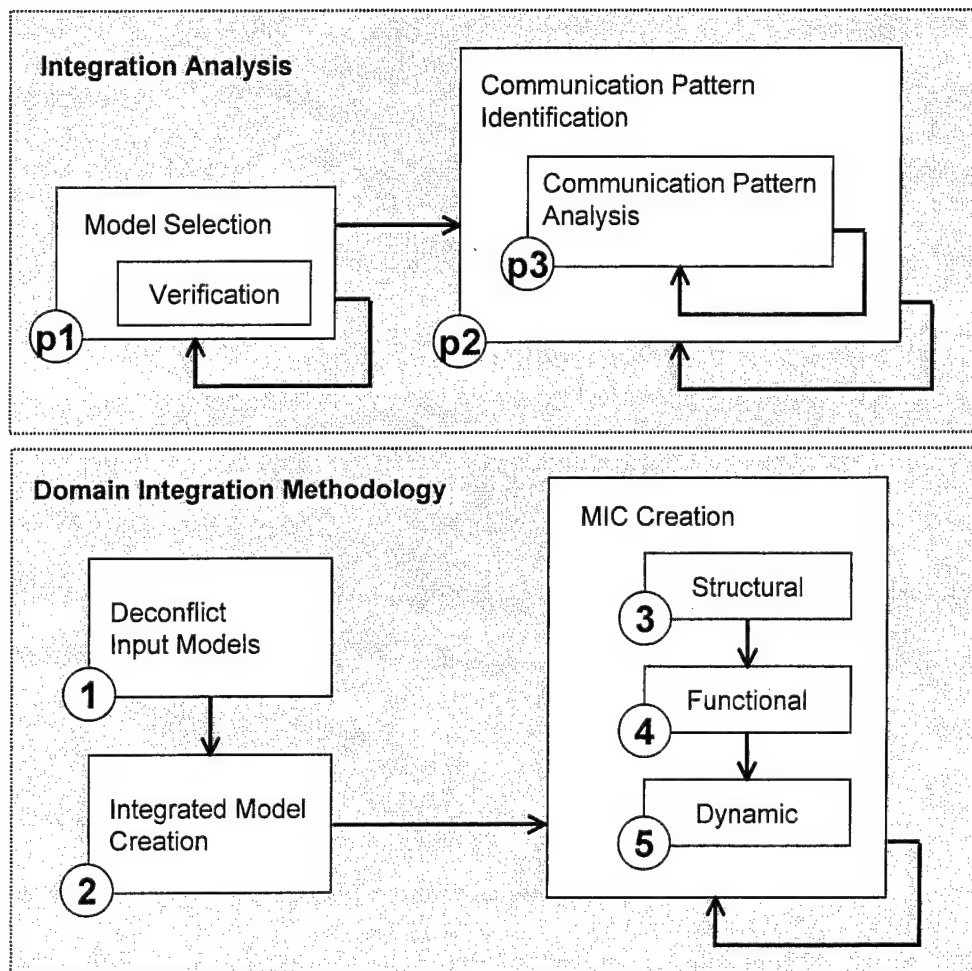


Figure 28. Domain Model Integration Methodology

4.2.1. *Deconflict the Input Models (Step 1).* After the input models have been selected, they must be prepared for the integration. The preprocessing of the models is to ensure that all model-level identifiers are deconflicted. Model-level refers to those identifiers that are viewable at the model-level. In other words, internal class identifiers such as attributes and methods do not need to be deconflicted. The following items must be examined against the identifiers in their corresponding categories in the other input models: types, constants, global subprograms, classes, aggregations, associations, and events. Note that events are the only exception to the model-level visibility rule because of their special rule outside of individual classes. Figure 29 and Figure 30 show two example input models being subjected to identifier deconfliction.

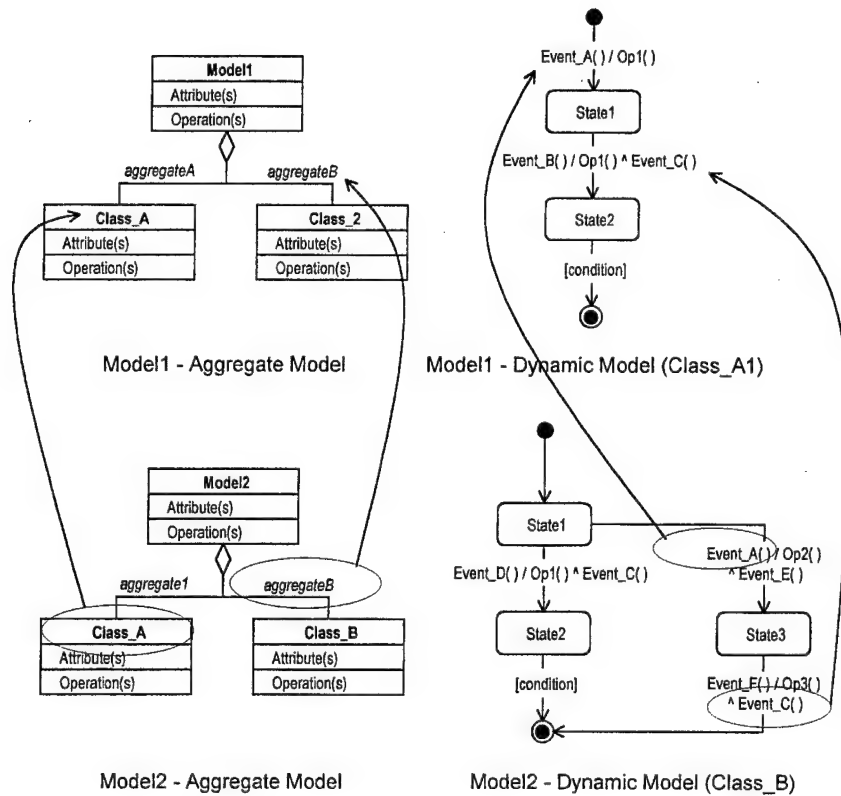


Figure 29. Input Models 1 and 2 Prior to Identifier Deconfliction

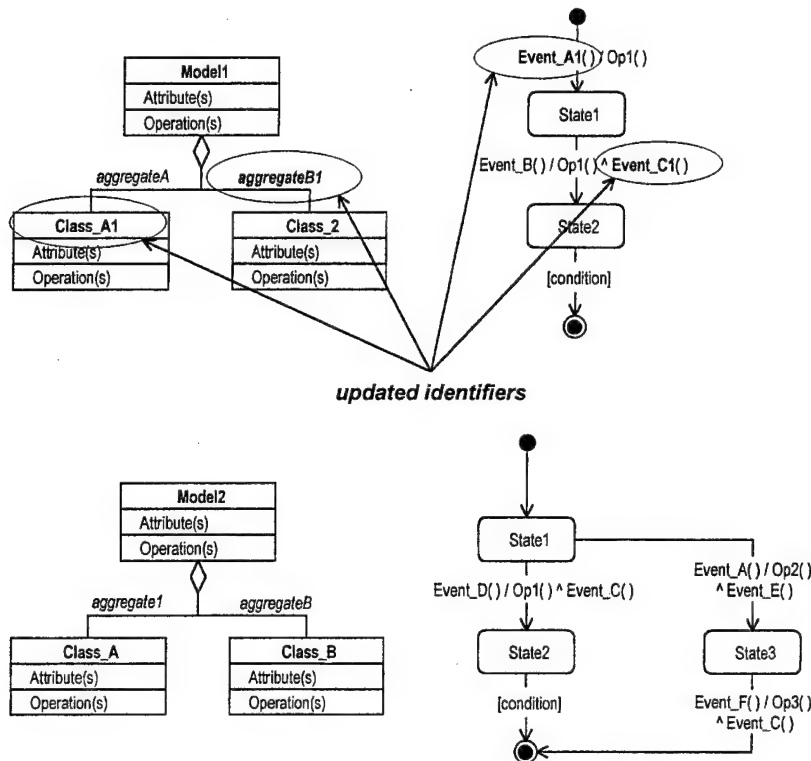


Figure 30. Input Models 1 and 2 Post-Deconfliction

4.2.2. *Create the New Integrated Model (Step 2).* Next, the new integrated model is created. This step is really composed of three smaller steps: creating the new model, creating a new system class, and creating the aggregate relationships from the input model system classes to the new system class of the integrated model. Figure 31 shows the end result of this step.

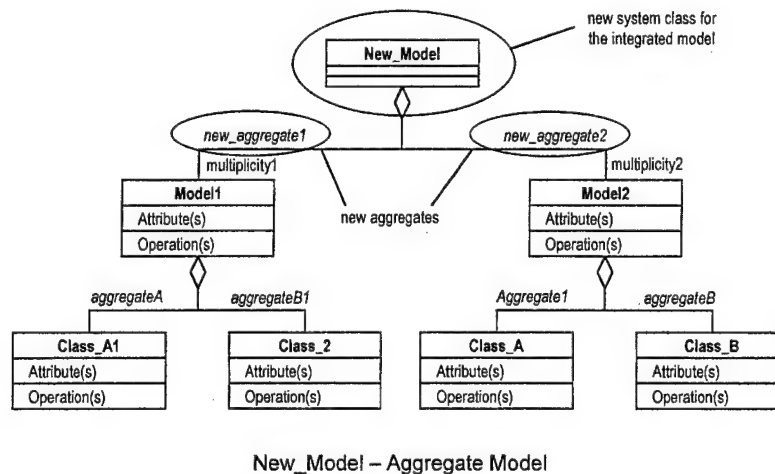


Figure 31. New Integrated Model (input models: Model1 and Model2)

4.2.3. *Create each MIC's Structural Component (Step 3).* By the end of this step, a new MIC is created for each of the identified communication patterns. At this point in the methodology, only the class and each of its attributes will be created. To facilitate the creation of the attributes, new type declarations need to be created – container types for each of the incoming events' parameters. Additionally, for each new MIC a corresponding aggregate relationship must be created to tie it to the new model's system class. Figure 32 is an example of adding a MIC to the integrated model. *New_MIC_Aggregate* is the aggregate that connects the class to *New_Model*. Additionally, the class *MIC* contains attributes for both the receive event parameters and the trigger strategy(ies).

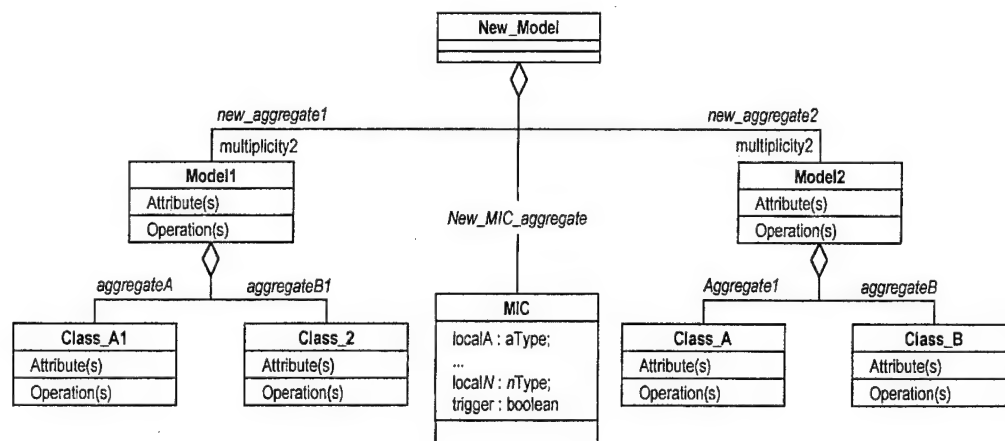


Figure 32. MIC Created for a Communication Pattern Example

4.2.4. *Create each MIC's Functional Component (Step 4).* With this step, the functional model is constructed for each new MIC. Each receive event must have its own process receive event procedure, and each must have its own evaluate trigger function. Additionally, each set of send events that are generated requires a separate conversion procedure. Each conversion procedure, in turn, must have a unique convert function for every parameter in the outgoing events. Finally, each conversion procedure needs its own initialization procedure.

Figure 33 provides a sample communication pattern in order to show the MIC's necessary subprograms: *processEvent_Y(a)*, *evaluateTrigger()*, *initializeTrigger()*, *convertBEvent_X()*, and *conversion(b)*.

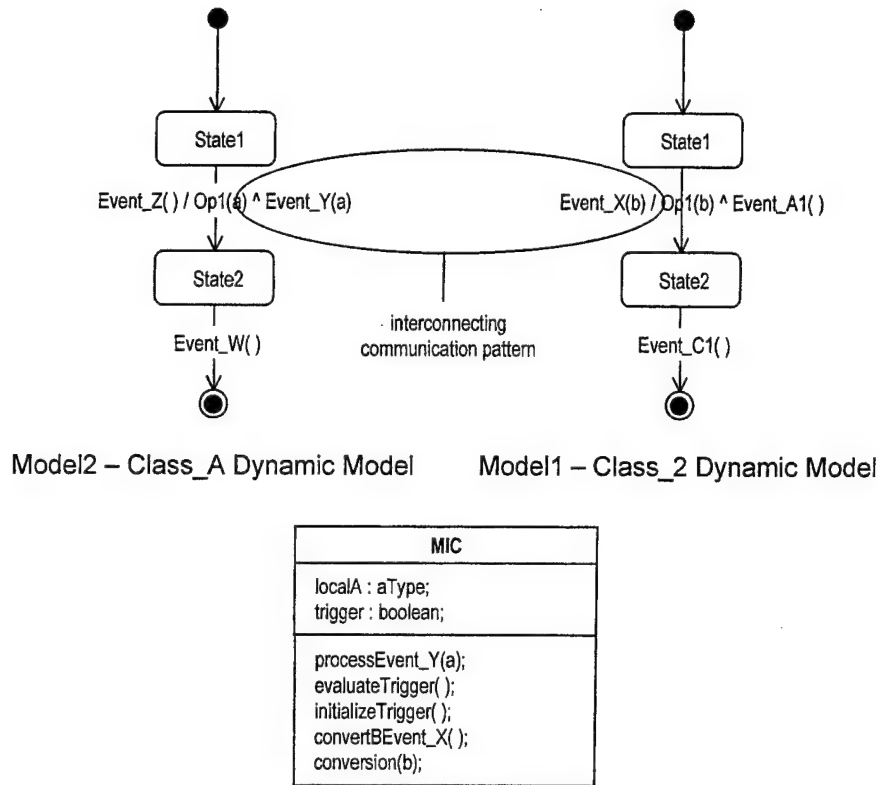


Figure 33. Example MIC's Functional Component from a sample communication pattern

4.2.5. *Create each MIC's Dynamic Component (Step 5).* With this step, the dynamic model for each new MIC is constructed. Once all of the methods have been created for a MIC, they must be combined with the receive and send events, guard conditions, and necessary states. The states *ReceiveEvents* and *ConversionDone* are required as a minimum. However, for each group of send events that a MIC produces, an additional set of these states are required. The diagram in Figure 34 shows an example dynamic model for a simple transfer communication pattern.

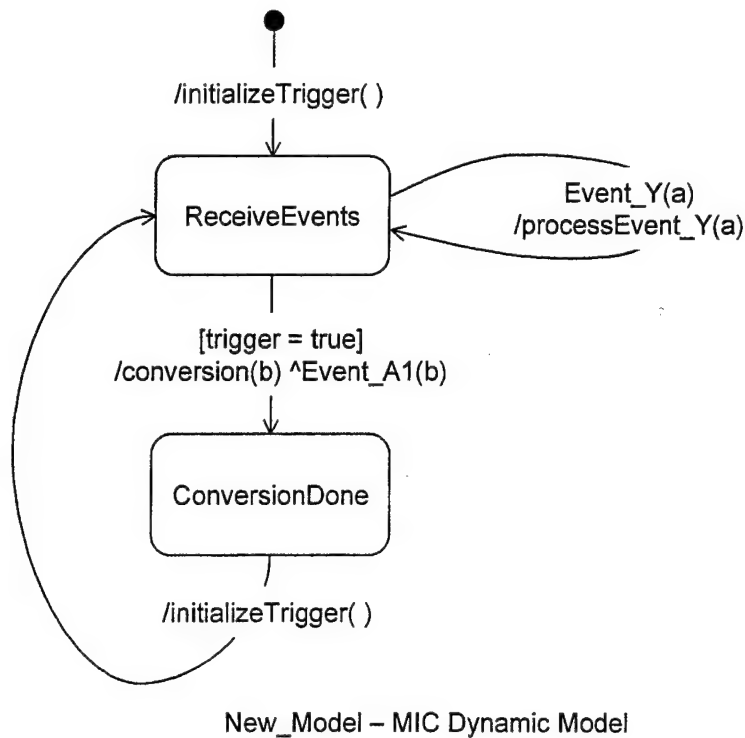


Figure 34. Example MIC's Dynamic Model

4.3. *AWL Specific Domain Integration Methodology*

Before presenting the AWL domain model integration methodology, a short summary of the steps is provided in order to help illustrate the commonality between the two methodologies. Each of the steps are represented in the following AWL methodology, and can be directly traced back to their corresponding steps in the UML methodology.

- Pre-Integration Step p1. Input selected domain models (*must be well-formed*)
- Pre-Integration Step p2. Communication Pattern Identification
- Pre-Integration Step p3. Communication Pattern Analysis
- Step 1. Deconflict the identifiers between the input domain models
- Step 2. Create the integrated model (*new model with input models merged*)
- Step 3. Create a new MIC for each communication pattern (*new class and attributes*)

- Step 4. Provide each MIC with its functional component (*subprograms*)
- Step 5. Provide each MIC with a dynamic model (*events, states, and transition table*)

Figure 28 illustrates the preceding steps and provides some insight into what portions of the process can be automated, specifically, those areas where user input is not required.

Automating steps 1-5, can further remove the application engineer from the burden of having detailed domain knowledge of the input models. Additionally, the automation eliminates the responsibility for the “hands-on” creation of the integrated mode. The same automated support tool can also assist the engineer during the pre-integration steps, by providing the pertinent model information needed for identifying and analyzing the necessary communication patterns.

Sections 4.3.1 through 4.3.8 provide a detailed explanation including examples of each of the seven integration steps that comprise the methodology.

4.3.1. Input Model Selection (Pre-Integration Step p1).

1. Select the input domain models for the integration. (see Section 3.1.1)
2. Verify that each model adheres to the well-formed domain rules. (see Section 2.2)

4.3.2. Communication Pattern Identification (Pre-Integration Step p2).

1. Identify all of the inter model communication patterns that satisfy requirements presented in the problem statement (see Section 3.2.3). For each, determine its communication pattern type (transfer, split, merge, mixed, or combined)
2. If a communication pattern is combined, determine its communication pattern type for each of the receive – send cycles. Also, each cycle requires its own trigger strategy. All other communication patterns have only one trigger strategy.
3. For each trigger strategy, identify its receive and send events.

4.3.3. Communication Pattern Analysis (Pre-Integration Step p3).

1. Analyze each trigger strategy to determine the strategy type (counter-based, value-based, or event-based).
 - 1.1. For counter-based strategies, determine the target count to trigger the send event(s).
 - 1.2. For value-based strategies, design a post-condition that triggers the send event(s). The expression is limited to use logical variables made available by the MIC's receive event parameters. Additionally, the expression must be correct with regard to type range restrictions and type bridging rules (see Sections 2.4.1 and 2.4.2).
 - 1.3. For event-based strategies, determine which receive event triggers the send event(s).
2. For every parameter in all of the communication pattern's send events, design a post-condition that generates the required value. The expression is limited to use logical variables made available by the MIC's receive event parameters. Additionally, the expression must be correct with regard to type range restrictions and type bridging rules (see Sections 2.4.1 and 2.4.2).

4.3.4. Deconflict the Input Domain Models (Step 1).

1. Compare the class identifiers between all input models. For each pair of non-unique identifiers, perform steps 1.1 and 1.2.
 - 1.1. Select one of the class identifiers and modify it by postpending a character to the identifier.

<pre>class ClassA is end class;</pre>	\Rightarrow	<pre>class ClassA1 is end class;</pre>
---------------------------------------	---------------	--

- 1.2. In the affected model, update all references to the modified identifier. Class identifiers can be referenced in the model's associations and aggregations.
2. Compare the events between all input models. If there are any duplicated event identifiers between models, perform steps 2.1 and 2.2.
 - 2.1. Select one of the event identifiers and modify it by postpending a character to the identifier.

<code>event EventA(); ⇒ event EventA1();</code>

- 2.2. In the affected model, update all references to the modified identifier. Event identifiers can be referenced in the model's class dynamic models, including its transition tables, and intra-model event associations.
3. Compare the identifiers for global constants, global subprograms, associations, and aggregations between all input models. For each pair of non-unique identifiers, perform steps 3.1 and 3.2.
 - 3.1. Select one of the identifiers and modify it by postpending a character to the identifier.

<code>MAX : constant NATURAL := 10 ⇒ MAX1 : constant NATURAL := 10 function Incr(x) : NATURAL ⇒ function Incr1(x) : NATURAL association RelA is ⇒ association RelA1 is aggregation HasClass is ⇒ aggregation HasClass1 is</code>

- 3.2. In the affected model, update all references to the modified identifier. These categories of identifiers can be found in the expressions of subprogram pre- and post-conditions, and class or event invariants.
4. Compare the type declarations between all input models. If there are any duplicate declarations, both the identifier and the definition, select one of the declarations and delete it.
5. Compare the remaining type declarations between all input models. For each pair of non-unique identifiers, perform steps 5.1 and 5.2.

- 5.1. Select one of the type identifiers and modify it by postpending a character to the identifier.

`MyInt : is range 1 .. 10; ⇒ MyInt1 : is range 1 .. 10;`

- 5.2. In the affected model, update all references to the modified identifier. Type identifiers can be referenced as parameter types in subprograms and events, other type declarations, constant declarations, and as types for class attributes.

4.3.5. *Create the New Integrated Model (Step 2).*

1. Create a new “empty” integrated model. Models are defined by packages in AWL.

`package NewSystem is
end package;`

2. In the new package, create the integrated model’s new system class.

`package NewSystem is
 class NewSystemClass is
 end class;
end package;`

3. Create a new aggregate relationship in the integrated model for each input model. In each aggregate, the parent is the newly created system class and the child is the system class from the input model. The relationship name is Has*, where * is the name of the input model’s system class. Lastly, the aggregation ends’ multiplicities need to be determined. The parent’s multiplicity is always one, but the child’s multiplicity is determined by the application engineer. If a suitable multiplicity type declaration does not exist, it must be created and added to the integrated model.


```

package NewSystem is
  type NewOneType is range 1 .. 1;
  type NewOneOrMoreType is range 1 .. *;

  class NewSystemClass is
    end class;

  aggregation HasInputModelSystemClass is
    parent p : NewSystemClass multiplicity NewOneType;
    child c : InputModelSystemClass multiplicity NewOneOrMoreType;
  end aggregation;
end package;

```

4. Copy all of the input models' package declarations to include: type and constant declarations, global subprograms, classes, associations, and aggregations as well as any nested package declarations, into the integrated model's package declaration. At this point, the input models can be discarded as they are no longer needed.

4.3.6. Create each MIC's Structural Component (Step 3).

1. For each communication pattern, perform steps 2 - 7.
2. Create a new class with the name **MIC(CommPatternIdent)** (where *CommPatternIdent* represents the identifier of the current communication pattern).

```

class MIC1 is
  end class;

```

3. Create a new aggregate relationship named Has*, where * is the name of the new MIC. The parent class is the new system class, and the child class is the newly created MIC. The multiplicity for both the parent and child is one.

```

aggregation HasMIC1 is
  parent p : NewSystemClass multiplicity One;
  child c : MIC1 multiplicity One;
end aggregation;

```

4. For every event utilized in the communication pattern, one of the following actions must be accomplished. If the event is an inter-model event, perform step 4.1, otherwise perform step 4.2. There is one other type of event connection pattern that must be addressed. If the event is a severed intra-model event perform step 4.3.

- 4.1. Create a new association relationship between the MIC and the supplying or receiving class. Note, the association's roles depend on whether the event is received or sent. Also the multiplicity for the association ends are always One.

When this step is complete, the event is now an intra-model event.

```
association EventName1 is
  role s : SendingClass multiplicity One;
  role r : MIC1 multiplicity One;
end association;

association EventName2 is
  role s : MIC1 multiplicity One;
  role r : ReceivingClass multiplicity One;
end association;
```

- 4.2. Modify the existing event association to include the MIC as either a new sender or receiver as necessary.

```
association EventName3 is
  role s : SendingClass multiplicity One;
  role r : ReceivingClass multiplicity One;
  role r1 : MIC1 multiplicity One;
end association;

association EventName4 is
  role s : SendingClass multiplicity One;
  role s1 : MIC1 multiplicity One;
  role r : ReceivingClass multiplicity One;
end association;
```

- 4.3. Delete the existing event association. Then treat the event as a new inter-model event by performing step 4.1. Note, once the event association is deleted, the event becomes an inter-model event for all future MIC's.
5. Repeat this step for every parameter in every receive event identified in the communication pattern, including all trigger strategies.
- 5.1. A container of the parameter type must exist in the integrated model. If one does not already exist, create it.

```
receive event → EventA (a : in AType, b : in AType)

package NewSystem is
  type ATypeContainer is bag of AType;
  ...
```

- 5.2. Create a new local attribute, with the appropriate container type. The attribute name is the parameter name postpended with the event name to ensure that all attribute identifiers are unique. This naming convention also assists the application engineer in determining what each local attribute represents.

```
receive event → EventA (a : in AType, b : in AType)

class MIC1 is
  aEventA : ATypeContainer;
  bEventA : ATypeContainer;
end class;
```

6. Create a new trigger attribute for each trigger strategy in the communication pattern. The type of this attribute is BOOLEAN, which is predefined in AWL. In order to ensure unique identifiers, the trigger strategy identifier is postpended to the attribute.

```
class MIC1 is
  aEventA : ATypeContainer;
  bEventA : ATypeContainer;
  triggerTSIdent : BOOLEAN;
end class;
```

7. For each trigger strategy that is counter-based, create a new counter attribute. The attribute's type is NATURAL which may already exist, or may need to be declared. In order to ensure a unique identifier, the trigger strategy's identifier is postpended to the attribute.

```
class MIC1 is
  aEventA : ATypeContainer;
  bEventA : ATypeContainer;
  triggerTSIdent : BOOLEAN;
  counterTSIdent : NATURAL;
end class;
```

4.3.7. Create each MIC's Functional Component (Step 4).

1. For each MIC, perform steps 2 through 6.
2. For each trigger strategy, create an evaluate trigger function. In all cases, the function return type is BOOLEAN, as the function's purpose is to set the trigger

attribute. The function's name is Evaluate*, where * is the trigger strategy's identifier.

- If the trigger strategy is event-based, the post-condition guarantees to return true.

```
function EvaluateTSIdent( ) : BOOLEAN
  guarantees (EvaluateTSIdent = True)
```

- If the trigger strategy is counter-based, the function compares the current event count to the specified target value, and returns the determined boolean value.

```
function EvaluateTSIdent(count : in NATURAL) : BOOLEAN
  guarantees
    ((EvaluateTSIdent = True and count = targetCount) or
     (EvaluateTSIdent = False and count /= targetCount))
```

- If the trigger strategy is value-based, the function computes a value using the MIC's local attributes. The computed value is then compared to a specified target value. The application engineer is responsible for supplying the: target value (targetValue), comparison operator (comparisonOp), and the comparison expression (expression). The complimentary comparison operator ($\overline{\text{comparisonOp}}$) represents the opposite boolean comparison operator to the one supplied by the user (= vs. /=, < vs. >=, and <= vs. >). Note, the expression should be verified to ensure that it is well-formed and correct. (See Figure 23. Convert Function's Assignment Expression Creation Flowchart, for an example of creating a correctly typed expression)

general form:

```
function EvaluateTSIdent( ) : BOOLEAN
  guarantees quantified expression (local declarations)
    ((EvaluateTSIdent = True and targetValue comparisonOp expression) or
     (EvaluateTSIdent = False and targetValue  $\overline{\text{comparisonOp}}$  expression))
```

specific example:

```
function EvaluateTS1( ) : BOOLEAN
  guarantees exists (x1, x2 : AType)
    ((x1 in ATypeContainer and x2 in ATypeContainer) and
     ((EvaluateTS1 = True and 100 < x1 + x2) or
      (EvaluateTS1 = False and 100 >= x1 + x2)))
```

3. For each of the trigger strategy's receive events, create a process receive event procedure. The name of the procedure is Process*, where * is the name of the receive event. The procedure has two functions. First, it must store the incoming parameters into the MIC's local attributes and second, it must update the appropriate trigger attribute.

- 3.1. Create a post-condition that inserts all of the receive event's parameters into their appropriate local attributes.

```
receive event → REvent(a : in AType, b : in AType)

class MIC1 is
  aREvent : ATypeContainer;
  bREvent : ATypeContainer;

  procedure ProcessREvent(a : in AType, b : in AType)
    guarantees exists (x1, x2 : AType)
      ((x1 = a and x1 in aREvent) and (x2 = b and x2 in bREvent))
    ...
```

- 3.2. If the trigger strategy is value-based or the trigger strategy is event-based and the receive event equals the *targetEvent*, then add an assignment expression to the procedure's post-condition. The expression calls the trigger strategy's evaluation function to determine the value of the triggerTSIdent attribute.

```
procedure ProcessREvent(a : in AType, b : in AType)
  guarantees exists (x1, x2 : AType)
    ((x1 = a and x1 in aREvent) and
     (x2 = b and x2 in bREvent) and
     (triggerTSIdent = EvaluateTSIdent( )))
```

- 3.3. If the trigger strategy is counter-based, add two expressions to the procedure's post-condition. The first expression guarantees the trigger strategy's event counter is incremented. The second expression calls the trigger evaluation function to evaluate the trigger attribute.

```
procedure ProcessREvent(a : in AType, b : in AType)
  guarantees exists (x1, x2 : AType)
    ((x1 = a and x1 in aREvent) and
     (x2 = b and x2 in bREvent) and
     (counterTSIdent' = counterTSIdent + 1) and
     (triggerTSIdent = EvaluateTSIdent(counterTSIdent')))
```

4. For each parameter in the MIC's send events, create a convert parameter function.

The function name is **Convert***, where ***** is the parameter name concatenated with the name of the send event. These functions are responsible for deriving values for the MIC's send event parameters. The application engineer creates the conversion expression(s) by utilizing literal values and local MIC attributes. Verification and validation of the supplied expressions are left to the user. Unlike the other MIC subprograms, the required post-condition of the convert functions must be derived completely by the application engineer. Note the conversion expression's type must ultimately result in a match with **ParmType**, where **ParmType** is the type of the outgoing event parameter.

```
ParmType equals AType
function ConvertParmEvent( ) : ParmType
  guarantees (x1, x2 : AType)
    ((x1 in aREvent and x2 in bREvent) and
    (ConvertParmEvent = (x1 + x2) / 50))

Type Mapping
function ConvertParmEvent( ) : ParmType
  guarantees (x1 : AType)
    ((x1 in aREvent) and
    (x1 = 1  $\Rightarrow$  ConvertParmEvent = "Option1") and
    (x1 = 2  $\Rightarrow$  ConvertParmEvent = "Option2"))

Type Casting
function ConvertParmEvent( ) : ParmType
  guarantees (x1 : AType)
    ((x1 in aREvent) and (ConvertParmEvent = ParmType'(x1)))
```

5. For each trigger strategy create a conversion procedure. The procedure's name is **Conversion***, where ***** is the trigger strategy's name. The only responsibility of this procedure is to "gather-up" the parameters needed to generate the MIC's send event(s). This task is accomplished by adding the appropriate convert parameter function for each parameter in the post-condition of the procedure.

```
Send Events  $\rightarrow$  SEvent1(p : out PType) and SEvent2(q : out QType)

procedure ConversionTSIdent(p : out PType, q : out QType)
  guarantees (p = ConvertPSEvent1() and q = ConvertQSEvent2())
```

6. For each trigger strategy create an initialization procedure named *Initialize**, where * is the trigger strategy's name. It is the responsibility of this procedure to "reset" the MIC's attributes that are associated with this trigger strategy. In the case of the local attributes, resetting is accomplished by assigning the empty set to each attribute. In the case of the trigger attributes, resetting means assigning false to the attribute. The last category of attributes are the event counters. These are reset by assigning them the value 0.

```

procedure InitializeTSIdent( )
  guarantees (aREvent = {} and bREvent = {} and triggerTSIdent = false)

Counter-based
procedure InitializeTSIdent( )
  guarantees (aREvent = {} and bREvent = {} and
    triggerTSIdent = false and counterTSIdent = 0)

```

4.3.8. Create each MIC's Dynamic Component (Step 5).

1. Create the MIC's dynamic model.

```

class MIC1 is
  ...
  dynamic model is
    transition table is
    end transition table;
  end dynamic model;
end class;

```

2. Add the MIC's send and receive events to the dynamic model. At a minimum, all dynamic models will have **AUTO()** and at least one receive and one send event.

```

dynamic model is
  event AUTO( );
  event REvent(a : in AType, b : in AType);
  event SEvent1(p : out PType);
  event SEvent2(q : out QType);

  transition table is
  end transition table;
end dynamic model;

```

3. For every trigger strategy, add a receive events state, **ReceiveEventsTSIdent**, and a conversion done state, **ConversionDoneTSIdent**. All dynamic models have at least one set of these states. Additionally, they also each have a **START** state.

```

dynamic model is
  event AUTO( );
  event REvent(a : in AType, b : in AType);
  event SEvent1(p : out PType);
  event SEvent2(q : out QType);

  state START;
  state ReceiveEventsTS1;
  state ConversionDoneTS1;

  transition table is
    end transition table;
end dynamic model;

```

4. The last step in creating the dynamic model is to add the transitions. There are four types of transitions. The first AUTO transition moves the dynamic model to the first receive event state. The second transition type receives and processes receive events. The third type is an AUTO transition that is taken only if the trigger guard is tripped. It moves the dynamic model from the receiving event state(s) to the conversion done state(s), and in the process, the trigger strategy's send event(s) are generated. The last transition type is also AUTO, and moves the dynamic model from the conversion done state(s) back to a receiving event state. Additionally, this transition performs the trigger strategy's initialization action.

```

dynamic model is
  event AUTO( );
  event REvent(a : in AType, b : in AType);
  event SEvent1(p : out PType);
  event SEvent2(q : out QType);

  state START;
  state ReceiveEventsTS1;
  state ConversionDoneTS1;

  transition table is
    in START on AUTO do InitializeTS1 to ReceiveEventsTS1;
    in ReceiveEventsTS1 on REvent do ProcessREvent to ReceiveEventsTS1;
    in ReceiveEventsTS1 on AUTO if triggerTS1 = true do ConversionTS1
      send SEvent1, SEvent2 to ConversionDoneTS1;
    in ConversionDoneTS1 on AUTO do InitializeTS1 to ReceiveTS1;
  end transition table;
end dynamic model;

```


4.4. AWSOME Domain Model Integration Tool (ADMIT)

Automated or semi-automated tools show promise in being able to further remove the application engineer from the mundane, but necessary, low-level integration details. The ADMIT proof of concept tool was created in order to demonstrate the feasibility of such tools. There is no discussion of the actual code, as this section is not intended to provide the code solution to this problem. For that matter, there are many different systems that could have been designed to integrate models written in AWL, and many others could be developed for domain models specified in other formal languages. The following paragraph describes only the tool's high-level structure as well as some of its functionality.

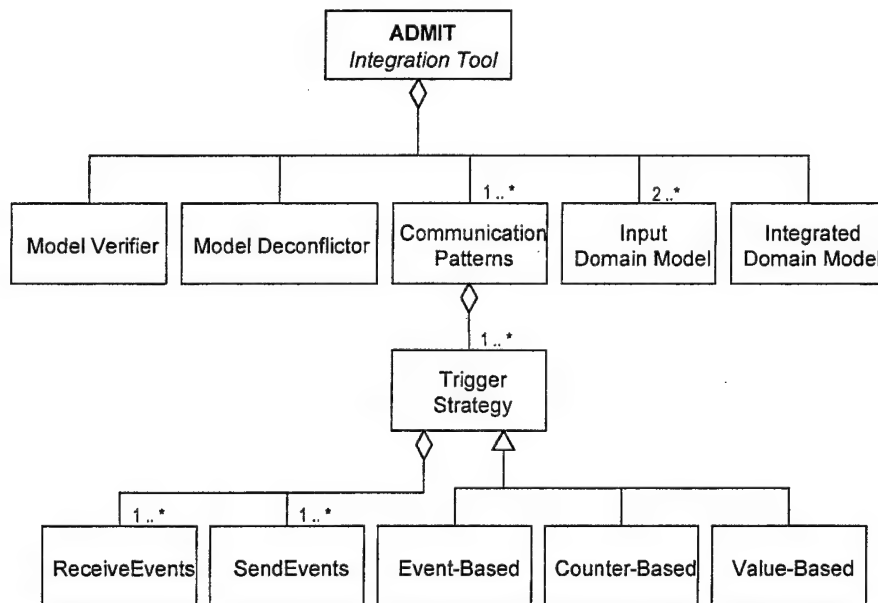


Figure 35. The ADMIT Object Model

As Figure 35 shows, ADMIT consists of a domain model verifier module, an input domain model deconflictor, and the integration tool itself. The verifier is responsible for determining whether or not a domain model follows the established rules to be a well-formed model. The deconflictor's responsibility is to ensure that an input set of domain models are processed such that each model's identifiers are unique to the identifiers of the other models.

Lastly, the main module, the integration tool, is responsible for ensuring that input domain models are entered, verified, and deconflicted. The tool then creates the integrated model, assists the application engineer with the integration analysis, and finally creates the required MICs, type declarations, associations, and aggregations. ADMIT was successful in accomplishing tool-assisted integrations of various input domain models. Refer to Section 5.1.3 (Secure Room Manager Integration Demonstration) for a run-time example of ADMIT. The output of the by-hand AWL integration and the tool-assisted integration were compared as a check to validate correctness. Additionally, the resulting models were subjected to the well-formed domain model verifier to ensure compliance with the established rules (Section 2.2).

5. AWL Domain Integration Methodology Demonstration

While Chapter 4 provides the reader with the general object-oriented and the derived AWL domain model integration methodologies, this chapter is intended to demonstrate their correctness. In order to accomplish this goal Section 5.1 shows the by-hand integration of two AWL input domain models, Room Manager System and Security Manager System. The models are provided in their entirety in Appendices C and D respectively, and the resulting integrated Secure Room Manager System is presented during the demonstration. In addition to the complete integration of the Secure Room Manager system, sample problems are provided in Section 5.2 to demonstrate how other types of communication patterns would be handled. Finally, Section 5.3 is provided to satisfy the goal of demonstrating how multi-agent systems are addressed by this methodology.

5.1. Security Manager and Room Manager Demonstration

Chapter 4 presented both the generic UML and the language specific AWL integration methodologies. Now, in order to demonstrate that the application of this approach produces a correctly integrated domain model, two sample input domain models will be subjected to the AWL integration methodology. The following sections walk the reader through the by-hand integration of the Security Manager and the Room Manager systems. The discussion is divided into the two portions as identified in Chapter 4.

5.1.1. Integration Analysis. Before the integration process can begin, an integration analysis must occur. As described earlier, the input models must be selected and analyzed to determine how they can be combined to satisfy the problem requirements.

5.1.1.1. *Input Model Selection (Pre-Integration Step p1).* The first step in the integration process is to select valid input models. In order to do this, the problem statement, as shown in Figure 36, is analyzed to determine what portions can be satisfied by domain models currently at hand. In this case, the requirements dealing with maintaining and viewing rooms can be satisfied by using the Room Manager System. Likewise, the requirements pertaining to security, system access and operation privileges are handled by the Security Manager System. If the available models did not address every problem statement requirement, the integration could still continue. The application engineer would then need to create the specifications for the missing requirement(s). However, this is not the case in this example. The *Integration Overview* worksheet is used to document the problem statement as well as the selected input models, including the breakdown as to which requirements are satisfied by each model. An additional concern in this step is the verification that the input domain models adhere to the well-formed domain model rules. As can be determined by examining the models, both the security and room manager systems conform to these rules. This information is also documented on the *Integration Overview* worksheet.

5.1.1.2. *Communication Pattern Identification (Pre-Integration Step p2).* At this point in the methodology, the input models have been selected - Room and Security Managers - and both have been verified to be well-formed. The next step is to identify and conduct an analysis of the communication patterns that will provide the integrated model with the functionality gained by combining the input models. This is accomplished by studying the events that each model produces or requires, and determining which events can be combined in order to

Domain Model Integration Worksheet Phase 1 (Integration Overview)	
Project Identification:	Secure Room Manager
Problem Statement: Create a secure room management system. The new system must allow the user to ¹ add rooms to the maintained set of rooms, and to ² view the set's contents by either a specific room or by rooms that meet a specified capacity. The security must ³ prevent unauthorized users from accessing the application, and ⁴ must ensure that only users with authorized privileges can perform specific application operations.	
Input Model #1:	Room Manager System
Model description: (The model is well-formed: <input checked="" type="checkbox"/> / N) The Room Manager system maintains a set of rooms from which queries can be made. The room has a name and building number, and is extended into the RoomWithCapy, which adds the room's seating capacity. The user is allowed to query the room set for specific rooms or can view a returned set of rooms that satisfy a given capacity.	
Problem statement requirement satisfied: Items 1 and 2.	
Input Model #2:	Security Manager System
Model description: (The model is well-formed: <input checked="" type="checkbox"/> / N) The Security Manager System is designed to be integrated with other application domain models. It was created to demonstrate the ability to introduce a security protocol to applications that were created with no inherent security. In addition to this goal, an additional purpose for this model was to have a suitable domain model for integration with other models that were created independent of this model. This model supplies security in two fashions. In the first, users are required to "log-in" prior to being granted access to the protected application. The second verifies the user's privilege before each application request to ensure the proper authorization. As a necessary aside, the system must also allow for the administrator to unlock users, and to change the user profiles. In order to implement the above goals, the Security System has three managers: AccessManager (AccMgr), ApplicationManager (AppMgr), and AdministrationManager (AdmMgr). Additionally, the system requires the use of a repository that contains the following tables: Session, Log, UserPasswords, UserRoles, UserAttempts, and RolePrivileges. The tables allow the various managers to view and update the current situation of any user.	
Problem statement requirement satisfied: Items 3 and 4.	

Figure 36. Secure Room Manager Input Model Selection Worksheet

share the desired information. Additionally, this step is responsible for identifying the trigger strategy(ies) that determines when the MIC can generate its send event(s). The starting point for this analysis is the interface contracts provided by each model. In this case, the inter-model event *MenuChoice* from Room Manager was identified as receiving the user's desire to perform one of that system's operations. In order for the Security Manager to ensure that only authorized users perform specific operations, that event should be "intercepted". Likewise, the Security Manager provides an event *DoAppReq*, also an inter-model event, which tells the protected application that

a valid operation request should be honored. Figure 37 shows the analysis that was performed to share the application request information between the security and room manager models. No other communication pattern was identified. Notice, however, that the other requirements, as specified by the problem statement, are implicitly satisfied by the integration of these models. Namely, the requirement that users must be “logged on” in order to access the application is satisfied because only logged on users can generate the necessary *DoAppReq* event. The worksheet in Figure 38 represents the application engineer’s analysis of the communication pattern identified in Figure 37.

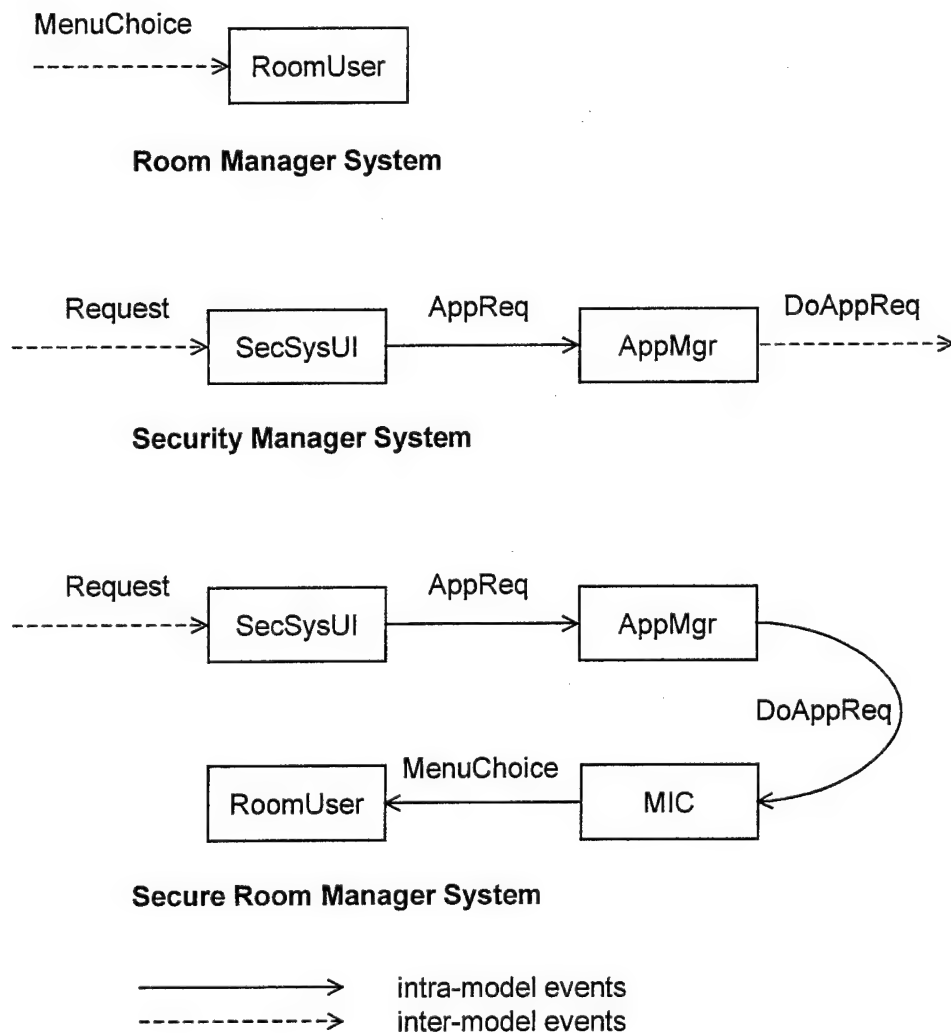


Figure 37. The Integration Plan for the Room and Security Models

Domain Model Integration Worksheet Phase 2 (Communication Pattern Identification)		
Interface Identification: MIC1		
What are the involved input models? Room Manager & Security Manager		
What is the information shared or purpose of this interface?		
<p>The user, responding to the Room Manager's menu prompt, sends a Request message to the Security Manager. The Security Manager then processes the request and submits a "DoAppReq" message to the protected application if the request is validated.</p> <p>The DoAppReq must be translated into the application's input message: MenuChoice.</p>		
What is the communication pattern (circle one)?		
<u>transfer</u>	split	merge
	mixed	combined
Sketch the communication pattern for the integration interface...		
<pre> sequenceDiagram participant User participant MIC User->>MIC: DoAppReq MIC->>App: MenuChoice </pre>		
Model	Event	Inter, Intra, or Severed
Security Manager System	DoAppReq	Inter (AppMgr – sender)
Room Manager System	MenuChoice	Inter (RoomUser – receiver)
Event	Parameter	Parameter Type
DoAppReq	op	STRING
MenuChoice	choice	MENUCHOICE

Figure 38. Secure Room Manager Communication Pattern Identification Worksheet

5.1.1.3. *Communication Pattern Analysis (Pre-Integration Step p3).* Now that the integration's one communication pattern has been identified, it must be further analyzed. The communication pattern, as shown in the preceding figure, falls in the transfer category, and as such requires only one trigger strategy. The strategy, cleverly named TS1, has one receive event, *DoAppReq*, and one send event, *MenuChoice*. In order to complete the analysis, however, several additional pieces of information must be gathered. Specifically, the trigger strategy must be determined. In this case, both the event-based and counter-based strategies would work equally well. However, in order to reduce the amount of input required by the application engineer, the event-based strategy was chosen because the event name is already known. If the counter-based strategy was chosen, the application engineer would be required to supply the information concerning how many occurrences of *DoAppReq* should be received prior to generating the send event. Another area of concern deals with analyzing how the parameters in the trigger strategy's

send event(s) are generated. In this case, *DoAppReq* has one parameter, *op*, of type *STRING*. This information, and the information about *MenuChoice*'s parameter (*choice* of type *MENUCHOICE*), is gathered by examining the models' interface contracts. Thus, the application engineer is now responsible for creating the correct conversion between *op* and *choice*. Because *STRING* and *MENUCHOICE* are both container types, the conversion type bridging *choice* is limited to type mapping. By studying Room Manager's interface contract, the valid input values for *MENUCHOICE* are determined to be: "add", "copy", "room", and "quit". This information can also be gathered by observing Room Manager's RoomUser dynamic model. The following expression was created by the application engineer to facilitate the conversion:

```
exists (x : STRING)
((x in opDoAppReq) and
 ( (x = "add" => ConvertChoiceMENUCHOICE = add) and
   (x = "copy" => ConvertChoiceMENUCHOICE = copy) and
   (x = "room" => ConvertChoiceMENUCHOICE = room) and
   (x = "quit" => ConvertChoiceMENUCHOICE = quit)))
```

Finally, one last concern must be addressed. As the inter-model events are connected to the corresponding MIC, they become intra-model events in the integrated model. Thus, when the MIC is created, the corresponding event-associations must also be created based on the class information listed in the interface contracts. The following worksheet, Figure 39, documents the choices made during step 4's integration analysis.

5.1.2. Domain Integration Methodology. Now that the analysis of the problem has been accomplished, the remaining integration steps are straightforward. In each of the methodology's steps, the information that is used to create portions of the integrated model can be traced back to the previously completed worksheets.

Domain Model Integration Worksheet Phase 3 (Communication Pattern Analysis)		
For each event the MIC receives...		
New Model Types (create a new container type for each unique parameter type): type STRINGContainer is bag of STRING;		
New MIC Attributes (create a local container attribute for each parameter in the receive event): private opDoAppReq : STRINGContainer;		
Each group of send events the MIC generates must have its own trigger strategy		
Trigger Strategy	Attributes	Evaluate Function
<input type="checkbox"/> counter-based	trigger countn	$(func = true \wedge count = target) \vee$ $(func = false \wedge count < target)$
<input checked="" type="checkbox"/> event-based	trigger	func = true
<input type="checkbox"/> value-based	trigger	$(func = true \wedge targetValue ? userExp) \vee$ $(func = false \wedge targetValue complimentary ? userExp)$
<input type="checkbox"/> other	trigger	supplied by the application engineer
For each parameter in the MIC's send event(s), create a Convert Function.		
Parameter	What is the logic (post-condition) that produces the desired output parameter from the available local attributes? (Consider both the type conversion as well as any value processing that needs to occur.)	
choice	exists (x : STRING) ((x in opDoAppReq) and ((x = "add" => ConvertChoiceMENUCHOICE = add) and (x = "copy" => ConvertChoiceMENUCHOICE = copy) and (x = "room" => ConvertChoiceMENUCHOICE = room) and (x = "quit" => ConvertChoiceMENUCHOICE = quit)))	

Figure 39. Secure Room Manager Communication Pattern Analysis Worksheet

5.1.2.1. *Deconflict the Input Domain Models (Step 1).* The first step of the methodology is to deconflict the input models. This ensures that after the integration, the newly created model does not have any ambiguity caused by non-unique identifiers of the same type. As is the case in the majority of these exercises, the two input models do not overlap in their use of identifier names. However, there is another aspect to model deconfliction. Both of the models have identical type declarations of: CHARACTER, STRING, NATURAL, One, and ZeroOrMore. These are not merely identifiers that need to be deconflicted, but rather they are duplicate declarations. To ensure that only one instance of each of these declarations makes its way into the integrated model, one occurrence of each is removed from its respective domain

model. No further modification of the input models is required by this change, as all references to the deleted declaration will still have a valid target once the input models have been merged together.

5.1.2.2. Create the New Integrated Model (Step 2). The second step is the creation of the new model. It requires input from the application engineer as to the name of both the new package and the package's new system class. After the creation of these items, all of the declarations of the input models are then copied into the new model. The last part of this step, creating the aggregation relationships, binds the input models into one aggregate model. The following AWL code results from the creation of the new package, system class, and aggregates. In the interest of space and readability, the input models' declarations (types, constants, global subprograms, classes, aggregates, and associations) are not shown.

```
package SecureRoomManager is
  class SecRoomSys is
    end class;

  aggregation HasRoomSys is
    parent p : SecRoomSys multiplicity One;
    child c : RoomSys multiplicity One;
  end aggregation;

  aggregation HasSecSys is
    parent p : SecRoomSys multiplicity One;
    child c : SecSys multiplicity One;
  end aggregation;

  // All of the Room Manager declarations
  // All of the Security Manager declarations

end package;
```

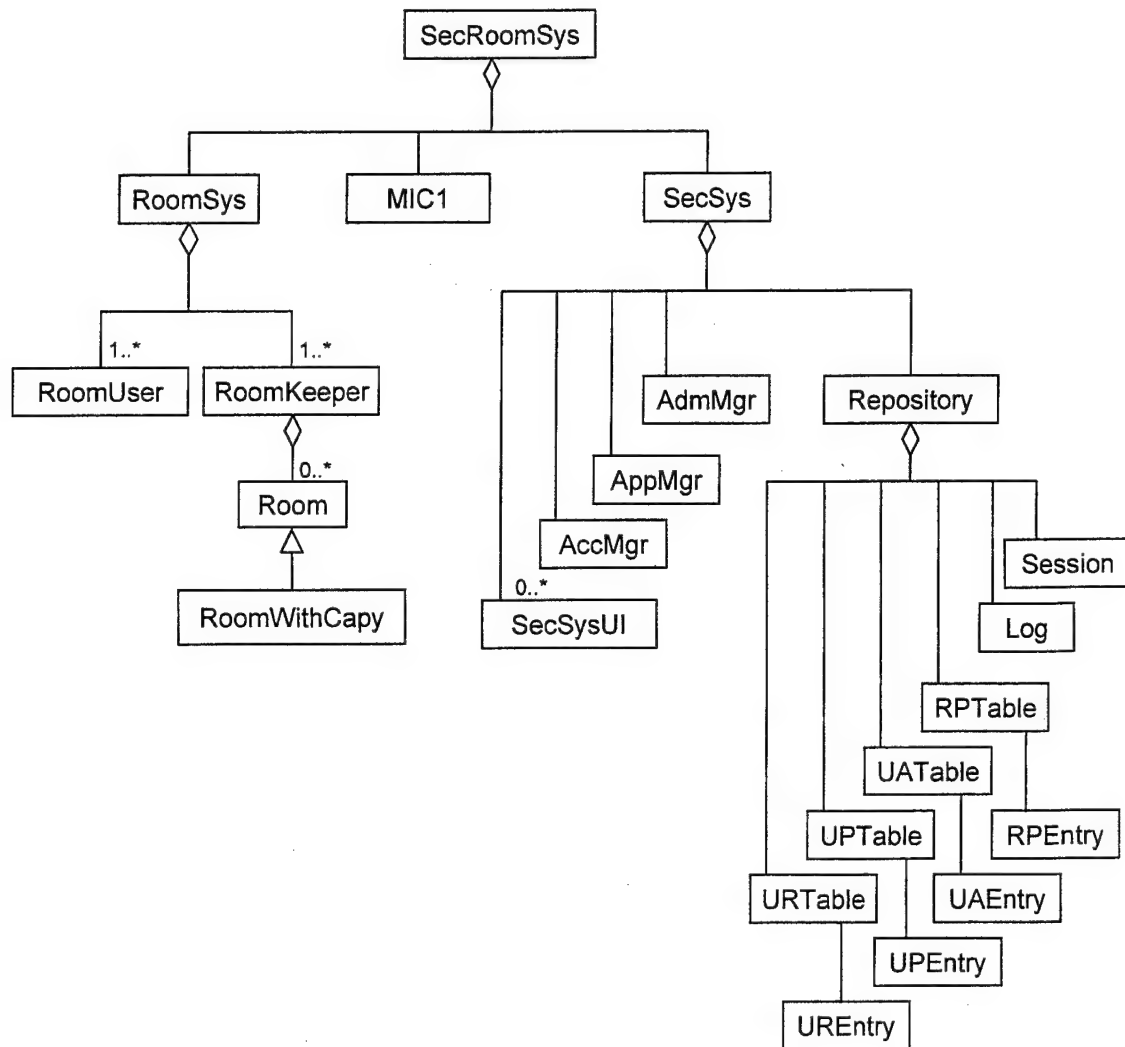


Figure 40. Integrated Secure Room Manager Aggregate Domain Model

5.1.2.3. *Create each MIC's Structural Component (Step 3).* Following the creation of the new model and the analysis of the integration communication patterns is the creation of the required MIC(s). Steps 3 - 5 all pertain to the completion of the MIC. This step handles the creation of each MIC and its attributes and completes the integrated model's aggregate model, as shown in Figure 40. In order to ensure that the possibly many MICs have unique identifiers, the communication pattern identifier is added to the end of "MIC" to create its name. The MIC's local attributes are derived by creating storage containers for each of the parameters in the communication pattern's receive events. In this example, the event *DoAppReq*

has one attribute, *op*, that requires a local attribute. Additionally, the trigger strategy requires its own trigger attribute. The step is completed by creating an aggregate relationship between the system class and the new MIC.

```

package SecureRoomManager is
  type STRINGContainer is bag of STRING;

  class SecRoomSys is
  end class;

  aggregation HasRoomSys is
    parent p : SecRoomSys multiplicity One;
    child c : RoomSys multiplicity One;
  end aggregation;

  aggregation HasSecSys is
    parent p : SecRoomSys multiplicity One;
    child c : SecSys multiplicity One;
  end aggregation;

  class MIC1 is
    private opDoAppReq : STRINGContainer;
    private triggerTS1 : BOOLEAN;
  end class;

  aggregation HasMIC1 is
    parent p : SecRoomSys multiplicity One;
    child c : MIC1 multiplicity One;
  end aggregation;
end package;

```

5.1.2.4. Create each MIC's Functional Component (Step 4). Creating the

MIC's functional component is a bit more demanding. There are five types of subprograms that are required. First, an evaluation function for the trigger strategy is created. In this case, the communication pattern is transfer and the trigger strategy is event-based, so the function always returns true. Second, a process receive event procedure is created for the *DoAppReq* event. The parameter *op* is stored into the local attribute that matches the parameter name concatenated with the event name. Also, because the event name matches the event name target, the post-condition must also call the newly created evaluation function. The third subprogram to be created is the convert function that prepares the outgoing parameter, *choice*. The expression identified during the integration analysis is inserted into the function's post-condition. The function's return type must match *choice*'s type, MENUCHOICE. The fourth type of method is the conversion

procedure. It is responsible for providing the necessary parameters for the MIC's send event(s). It accomplishes this task by calling the corresponding convert functions for its out parameters. The last type of class subprogram is the initialization procedure, which ensures that the MIC's two attributes, *triggerTS1* and *opDoAppReq*, are reset. Because there is only one trigger strategy, the MIC requires just one evaluation function, conversion procedure, and initialization procedure. Coincidentally, there is also only one process receive procedure because the communication pattern is transfer, and only one convert function because there is only one parameter in the MIC's only send event. Figure 41 represents the MIC after completing steps 4 and 5, the creation of the structural and functional components.

```

class MIC1 is
  private opDoAppReq : STRINGContainer;
  private triggerTS1 : BOOLEAN;

  private function EvaluateTS1() : BOOLEAN
    guarantees (EvaluateTS1 = true)

  private procedure ProcessDoAppReq(op : in STRING)
    guarantees (x1 : STRING)
    (x1 in opDoAppReq and x1 = op and triggerTS1 = EvaluateTS1())

  private function ConvertChoiceMenuChoice() : MENUCHOICE
    guarantees exists (x : STRING)
    ((x in opDoAppReq) and
     ((x = "add" => ConvertChoiceMENUCHOICE = add) and
      (x = "copy" => ConvertChoiceMENUCHOICE = copy) and
      (x = "room" => ConvertChoiceMENUCHOICE = room) and
      (x = "quit" => ConvertChoiceMENUCHOICE = quit)))

  private procedure ConversionTS1 (choice : out MENUCHOICE)
    guarantees (choice = ConvertChoiceMenuChoice())

  private procedure InitializeTS1()
    guarantees (opDoAppReq' = {} and triggerTS1' = false)
end class;

aggregation DoAppReq is
  role s : AppMgr multiplicity One;
  role r : MIC1 multiplicity One;
end aggregation;

aggregation MenuChoice is
  role s : MIC1 multiplicity One;
  role r : RoomUser multiplicity One;
end aggregation;

```

Figure 41. Secure Room Manager MIC (without the dynamic model)

5.1.2.5. *Create each MIC's Dynamic Component (Step 5).* The final step in creating the integrated model is to generate the MIC's dynamic component. Figure 42 and Figure 43 illustrates MIC1's dynamic model. No further analysis is needed to create the necessary states and transitions. The states are added based on the number of trigger strategies, only one in this case. Therefore, the only required states are: START (always mandatory), ReceiveEventsTS1, and ConversionDoneTS1. Additionally all of the MIC's events, both send and receive are known: AUTO (always mandatory), DoAppReq, and MenuChoice. By following the steps in Section 4.3.8 the following dynamic model is created. The state diagram in Figure 43 represents MIC1's dynamic model.

```
dynamic model is
  event AUTO( );
  event DoAppReq(op : in STRING);
  event MenuChoice(choice : out MENUCHOICE);

  state START;
  state ReceiveEventsTS1;
  state ConversionDoneTS1;

  transition table is
    in START on AUTO do InitializeTS1 to ReceiveEventsTS1;
    in ReceiveEventsTS1 on DoAppReq do ProcessDoAppReq to ReceiveEventTS1;
    in ReceiveEventsTS1 on AUTO if triggerTS1 = true do ConversionTS1
      send MenuChoice to ConversionDoneTS1;
    in ConversionDoneTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
  end transition table;
end dynamic model;
```

Figure 42. Dynamic Model for Secure Room Manager's MIC

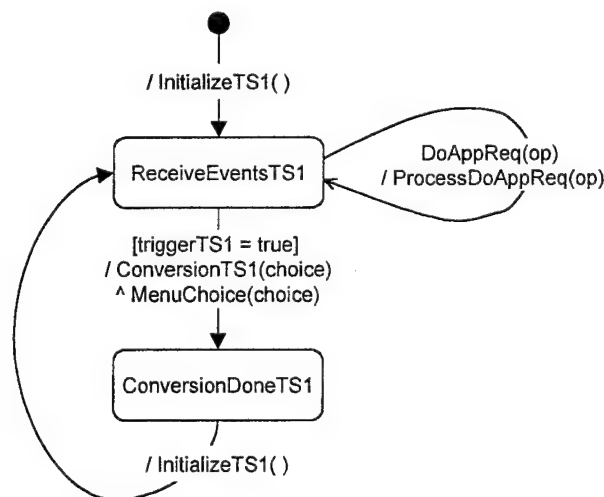


Figure 43. MIC1's Dynamic Model (Secure Room Manager)

The integration is now complete, with the exception of updating the new model's documentation. In addition to the integrated model's description, which should include each of the input models' descriptions, the following changes need to be made to the merged interface contract. The updated interface contract table reflects the change from inter- to intra-model on the events used to integrate the Room Manager and Security Manager.

Table 6. Secure Room Manger's Updated Interface Contract (only modified entries)

Event	Intra / Inter – Class (s / r)		Description
	Parameters	Parameter Type	
DoAppReq	Intra		Input to MIC1 from the input model: Security Manager System.
	op	STRING	
MenuChoice	Intra		Output from MIC1 to the input model: Room Manager System.
	choice	MENUCHOICE	

There are many different examples that could have been selected to demonstrate the workings of this methodology. However, while the Secure Room Manager System has a fairly straightforward transfer communication pattern, it shows the steps and decisions necessary to complete a successful integration of two input domain models. The succession from the generic methodology to the specific AWL methodology can be extended even further by using a software tool to assist the application engineer in completing the integration.

5.1.3. ADMIT Demonstration. As discussed in Section 4.4, ADMIT was developed to demonstrate the feasibility of automating this methodology. It was tested and validated on the Security Manager/Room Manager integration, the additional examples in Section 5.2, and the integration of many other domains. Figure 44 through Figure 49 represent the series of screen captures from the Secure Room Manager integration, Section 5.1.

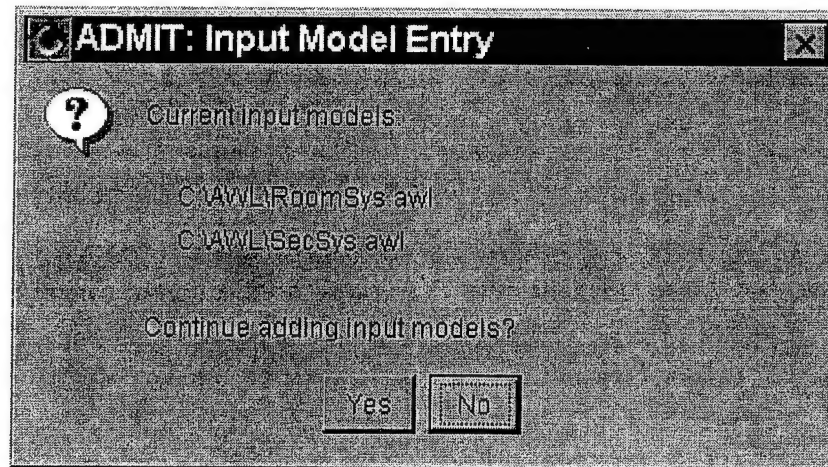


Figure 44. Pre-step 1: Input Model Selection (and verification)

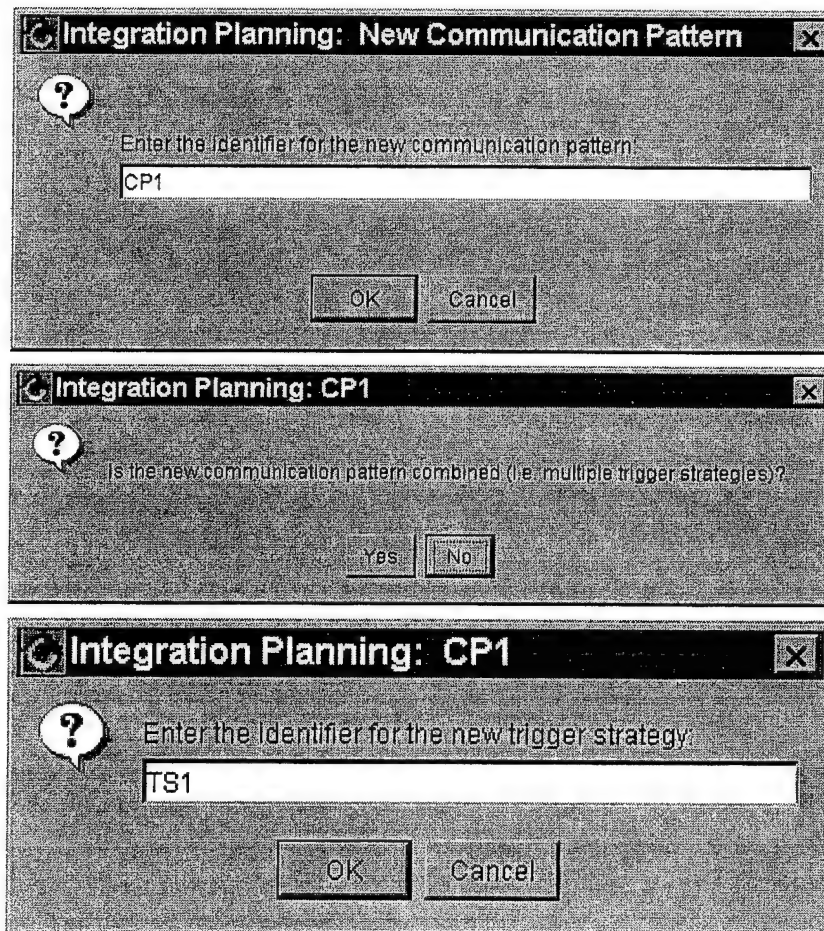


Figure 45. Pre-step 2: Communication Pattern Identification

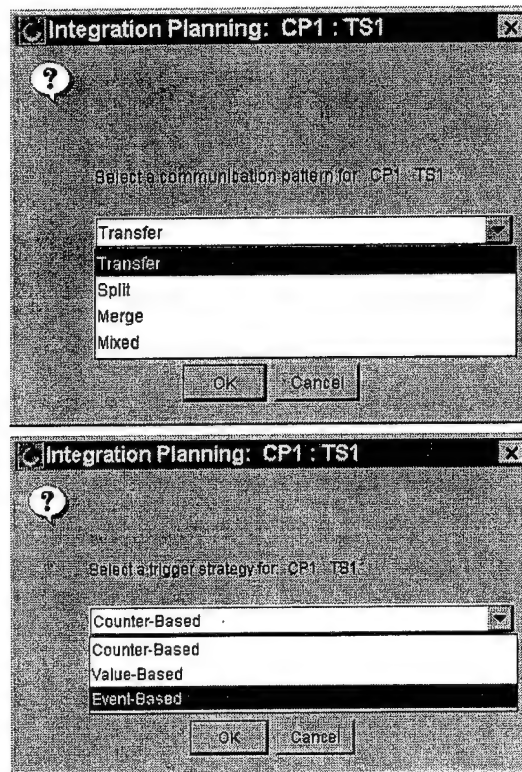


Figure 46. Pre-Step 3: Communication Pattern Analysis (*pattern recognition*)

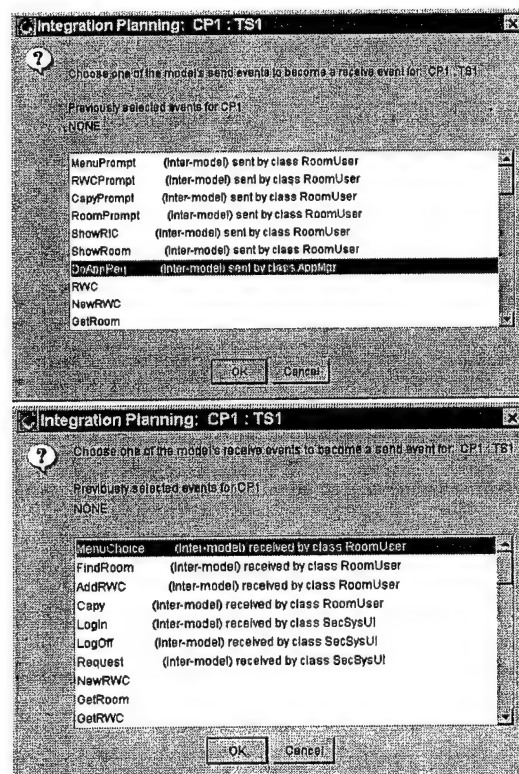


Figure 47. Pre-step 3: Communication Pattern Analysis (*event selection*)

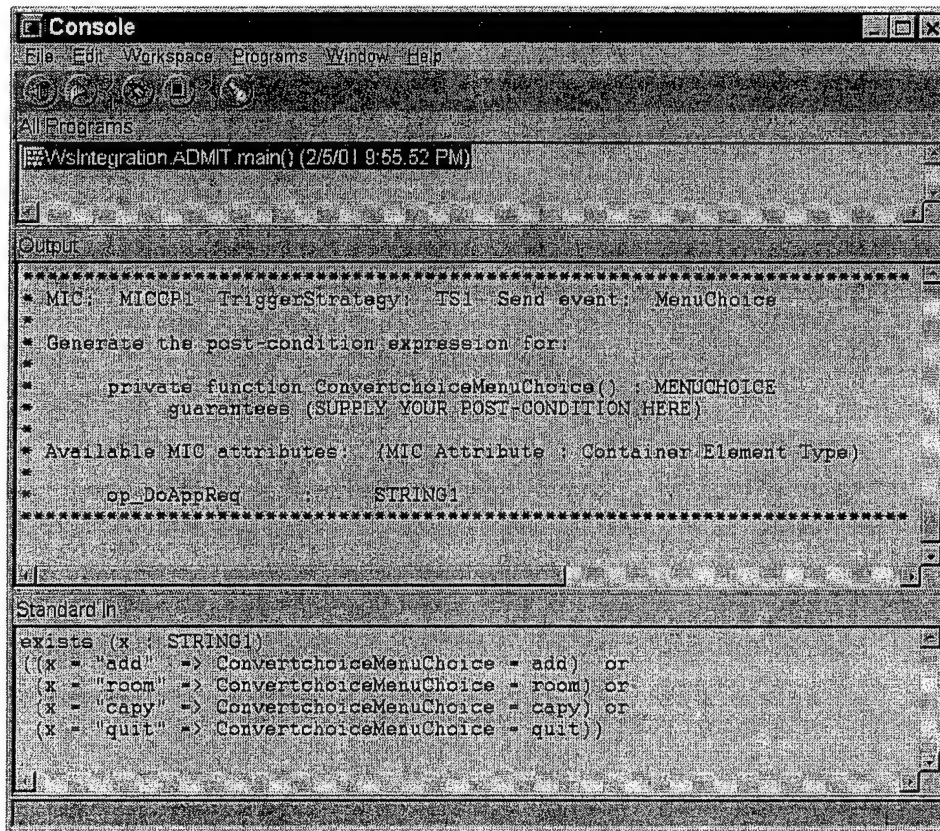


Figure 48. Pre-step 3: Communication Pattern Analysis (*conversion expression*)

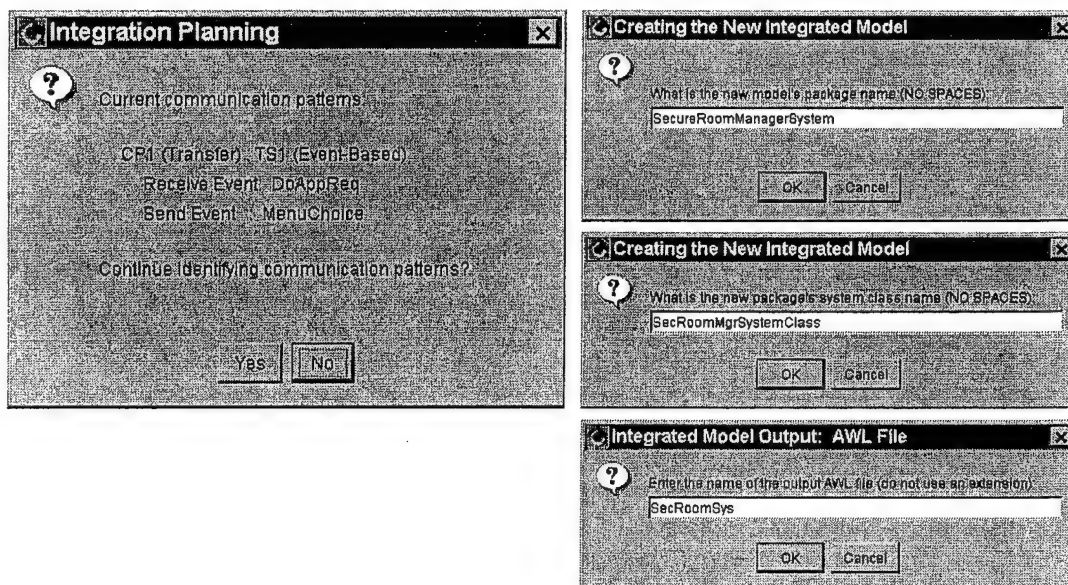


Figure 49. Automated Integrated Model and MIC Creation Steps

5.2. Additional Examples (non-simple communication patterns and conversions)

In order to demonstrate that the methodology handles other, more interesting possibilities, the following examples are provided to show how other integrations may occur. In the examples only the pertinent send and receive events from the input models are considered. This will allow the focus to be placed on the creation of the MICs. Additionally, only the MIC and any necessary type declarations are shown. Other package declarations are implied by the methodology.

5.2.1. Merge Communication Pattern (multiple attribute values). Unlike the Secure Room Manager demonstration, this example integration requires multiple receive events. Also, the generation of the send event's parameter requires the application engineer to supply a convert function that utilizes all of the values in the MIC's container attribute. Figure 50 depicts an example where multiple events, *TheScore*, are received before the single send event, *TheGrades*, is generated. Additionally, the type *ScoreType* is not directly compatible with *GradeType*, the elements of *GradeSet*.

The challenge in this scenario is first to receive the same event multiple times, storing each input value, then generate the send event by computing the converted values and packaging the information into a completely different value type. The communication pattern identified is *merge* and the trigger strategy is counter-based, where the target count is seven. That information, and the receive / send events, is enough to generate most of the MIC. The only remaining piece is the conversion function. The first part of the conversion must ensure that each input value is multiplied by two and converted from *ScoreType* to *GradeType*. Then, the converted values must be added to *GradeSet* in order to be output. The MIC generated by the methodology is shown in Figure 51.

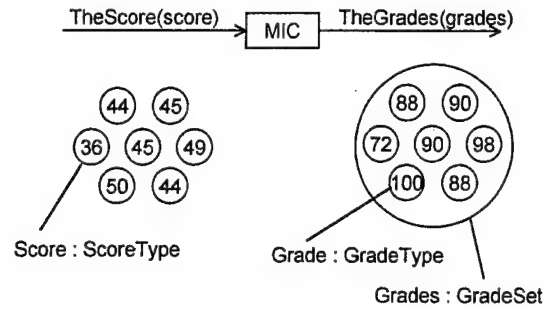


Figure 50. Merge Communication Pattern Example

```

...
type ScoreType is range 0 .. 50;
type GradeType is range 0 .. 100;
type GradeSet is set of GradeType;
type ScoreTypeContainer is bag of ScoreType;
...
class MIC is

    score_TheScore : ScoreTypeContainer;
    counterTS : NATURAL;
    triggerTS : BOOLEAN;

    private procedure ProcessTheScore(score : in ScoreType)
        guarantees (score in score_TheScore' and
            counterTS' = counterTS + 1 and
            triggerTS' = EvaluateTS(counterTS'))

    private function EvaluateTS(counter : in NATURAL) : BOOLEAN
        guarantees ((EvaluateTS = TRUE and counter = 7) or
            (EvaluateTS = FALSE and counter /= 7))

    private function ConvertGradesTheGrades() : GradeSet
        guarantees forall (x : ScoreType)
            (exists (y : GradeType, z : GradeSet)
                (x in score_TheScore and
                    y = x * 2 and
                    y in z and
                    ConvertGradesTheGrades' = z))

    private procedure ConversionTS(grades : out GradeSet)
        guarantees (grades' = ConvertGradesTheGrades())

    private procedure InitializeTS()
        guarantees (counterTS' = 0 and
            triggerTS = FALSE and
            score_TheScore' = {})

    dynamic model is
        event AUTO();
        event TheScore(score : in ScoreType);
        event TheGrades(grades : out GradeSet);

        state START;
        state ReceiveEvents;
        state ConversionDone;

        transition table is
            in START on AUTO do InitializeTS to ReceiveEvents;
            in ReceiveEvents on TheScore do ProcessTheScore to ReceiveEvents;
            in ReceiveEvents on AUTO if triggerTS = TRUE do ConversionTS
                send TheGrades to ConversionDone;
            in ConversionDone on AUTO do InitializeTS to ReceiveEvents;
        end transition table;
    end dynamic model;
end class;

```

Figure 51. MIC Generated from Merge Communication Pattern Example

5.2.2. *Combined Communication Pattern.* Another difficulty not addressed in the Secure Room Manager integration was dealing with the combined communication pattern. To demonstrate the creation of a MIC that has multiple trigger strategies, the example from Section 5.3.1 was modified to reflect the need to “prompt” the supplying model for additional information. Figure 52 shows the MIC requesting additional events before generating *TheGrades*. Refer to Figure 53 to visualize the dynamic model that represents the AWL code in Figure 54.

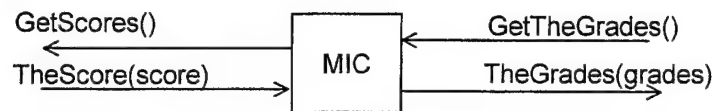


Figure 52. Combined Communication Pattern Example

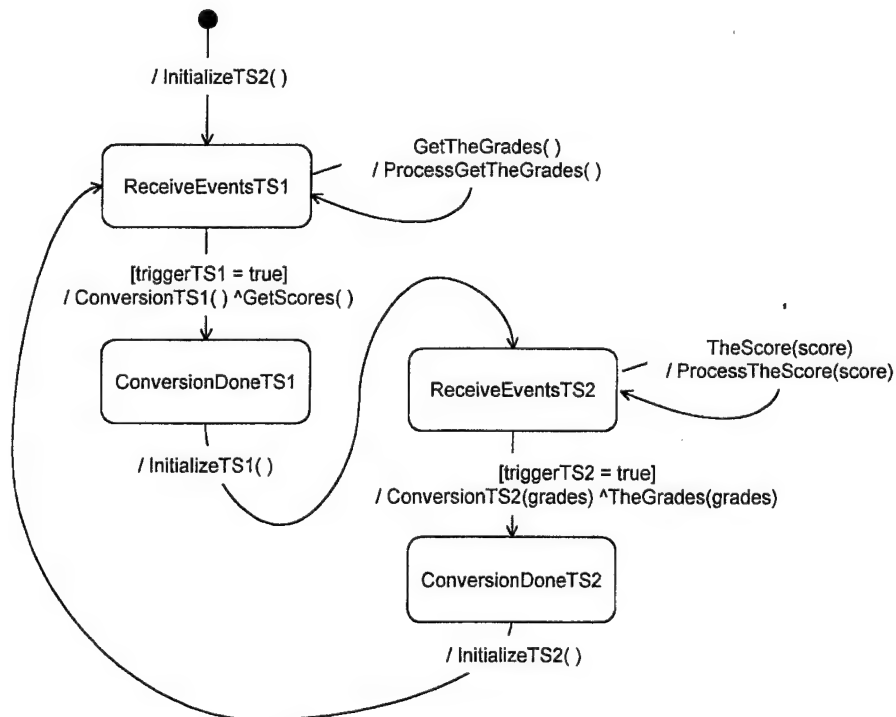


Figure 53. Combined Communication Pattern Example (Dynamic Model)

```

...
class MIC is

    score_TheScore : ScoreTypeContainer;
    counterTS2 : NATURAL;
    triggerTS1 : BOOLEAN;
    triggerTS2 : BOOLEAN;

    private procedure ProcessTheScore(score : in ScoreType)
        guarantees (score in score_TheScore' and
            counterTS2' = counterTS2 + 1 and
            triggerTS2' = EvaluateTS2(counterTS2'))

    private procedure ProcessGetTheGrades()
        guarantees (triggerTS1' = true)

    private function EvaluateTS1() : BOOLEAN
        guarantees (EvaluateTS1 = TRUE)

    private function EvaluateTS2(counter : in NATURAL) : BOOLEAN
        guarantees ((EvaluateTS2 = TRUE and counter = 7) or
            (EvaluateTS2 = FALSE and counter /= 7))

    private function ConvertGradesTheGrades() : GradeSet
        guarantees forall (x : ScoreType)
            (exists (y : GradeType, z : GradeSet)
                (x in score_TheScore and y = x * 2 and y in z and
                    ConvertGradesTheGrades' = z))

    private procedure ConversionTS1()

    private procedure ConversionTS2(grades : out GradeSet)
        guarantees (grades' = ConvertGradesTheGrades())

    private procedure InitializeTS1()
        guarantees (triggerTS1' = FALSE)

    private procedure InitializeTS2()
        guarantees (triggerTS2' = FALSE and counterTS2' = 0 and score_TheScore' = {} and
            triggerTS1' = FALSE)

    dynamic model is
        event AUTO();
        event TheScore(score : in ScoreType);
        event TheGrades(grades : out GradeSet);
        event GetTheGrades();
        event GetScores();

        state START;
        state ReceiveEventsTS1;
        state ReceiveEventsTS2;
        state ConversionDoneTS1;
        state ConversionDoneTS2;

        transition table is
            in START on AUTO do InitializeTS2 to ReceiveEventsTS1;
            in ReceiveEventsTS1 on GetTheGrades do ProcessGetTheGrades to ReceiveEventsTS1;
            in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1 send GetScores
                to ConversionDoneTS1;
            in ConversionDoneTS1 on AUTO do InitializeTS1 to ReceiveEventsTS2;
            in ReceiveEventsTS2 on TheScore do ProcessTheScore to ReceiveEventsTS2;
            in ReceiveEventsTS2 on AUTO if triggerTS2 = TRUE do ConversionTS2 send TheGrades
                to ConversionDoneTS2;
            in ConversionDoneTS2 on AUTO do InitializeTS2 to ReceiveEventsTS1;
        end transition table;
    end dynamic model;
end class;

```

Figure 54. MIC Generated from Section 5.2.2's Example

5.2.3. *Split Event Pattern.* Another possible situation that is interesting to explore is the split event pattern. Some integrations may require that the intra-model communication of a model be intercepted either to control the flow of processing or to modify the passed information. In either case, the methodology must recognize this pattern, and “break” the appropriate event association and create the replacements.

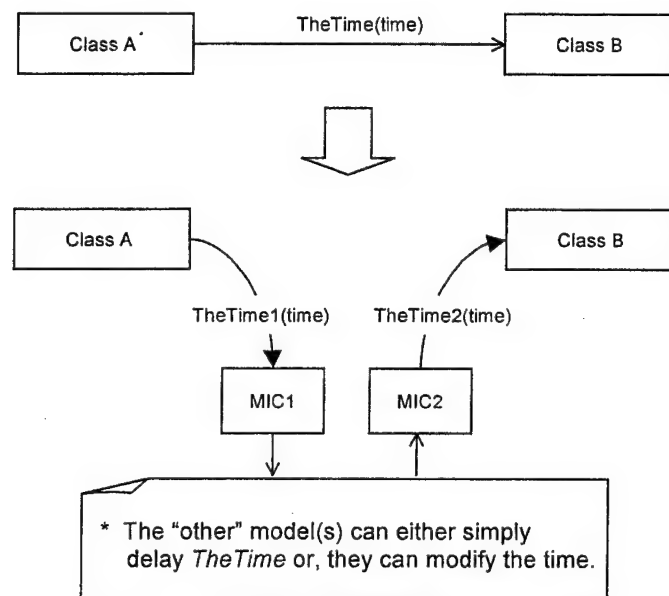


Figure 55. Split Event Pattern Example

```

// association from the input model...
association TheTime is
    role s : ClassA multiplicity One;
    role r : ClassB multiplicity One;
end association;

// associations in the integrated model
// WITHOUT association TheTime...

association TheTime1 is
    role s : ClassA multiplicity One;
    role r : MIC1 multiplicity One;
end association;

association TheTime2 is
    role s : MIC2 multiplicity One;
    role r : ClassB multiplicity One;
end association;

```

Figure 56. Integrated Model Associations Generated by a Split Event Pattern

5.3. Multi-Agent Domain Model Integration

One of the goals of this research was to demonstrate how existing models could be integrated with multi-agent systems. If successful, this technique could be used to update existing models in to fit into today's increasingly prevalent distributed multi-agent environments. While studying multi-agent systems, it became apparent that the original premise of using analysis level domain models excluded or limited some of the possible integrations that could occur. The demonstration of the integration between the Room Manager System and agentMom, a multi-agent system used at AFIT, illustrated this point. The methodology's limitation in this regard will be shown in the following discussion, and a possible solution is provided in Section 5.3.2.

5.3.1. Room Manager and agentMom Integration Demonstration. The multi-agent system chosen for the integration with Room Manager was agentMom. It is an implementation framework that defines how agents communicate in a distributed environment. In the framework, agents are equipped with a message handler that allows agents to receive messages that begin a new conversation. Each agent that participates in a conversion has a conversation half that is either the initiator or responder to the communication. A message class is used to implement events that pass between the two agents during the conversation. The messages, upon receipt, are parsed to extract their purpose (performative) as well as any content information contained in the communication. Unfortunately, this does not conform to the model definition assumed by this methodology.

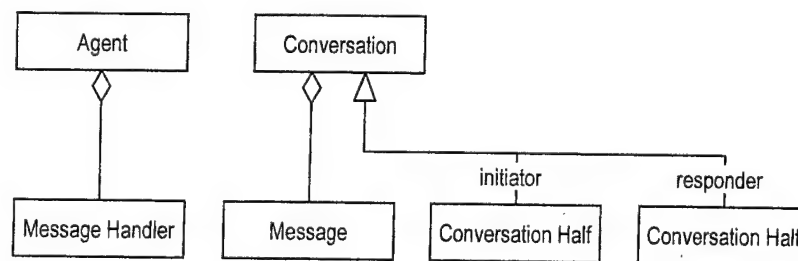


Figure 57. agentMom Object Model [14, 16]

As can be seen in Figure 57, the object model for agentMom does not fit the well-formed domain model rules as outlined in Section 2.2. While the model can be “fixed” by adding a top-level system class, there is another problem. The object model for agentMom is really a framework for implementing applications using a distributed message passing paradigm. It exists at the implementation rather than the analysis level of model representation needed by the integration methodology. Therefore, “integration” with existing systems such as Room Manager can be accomplished, but only in a ad hoc fashion by taking the analysis of the input model and using the framework provided by a multi-agent framework such as agentMom (see Figure 58).

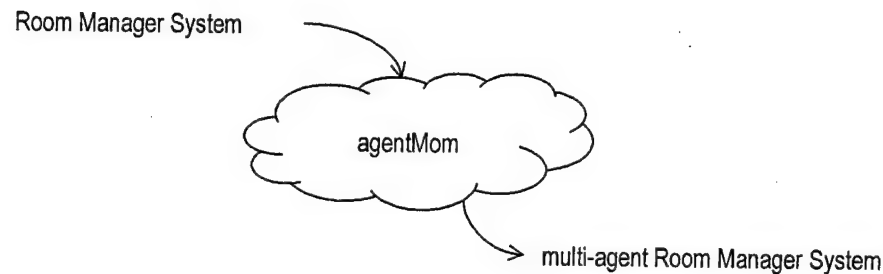


Figure 58. Multi-Agent System Creation Process Using agentMom



Figure 59. Room Manager System Message Passing Diagram (only two messages)

In order to create a multi-agent system from the Room Manager model, a pair of events were studied to provide an example implementation. Figure 59 shows the *RoomUser*, acting as the client, requesting a “RWC” from the *RoomKeeper*, acting as the server. Using agentMom’s object model, both *RoomUser* and *RoomKeeper* become agents, and each have a conversation half. The *RoomUserAgent*’s conversation becomes the initiator and the *RoomKeeperAgent*’s becomes the responder. Meanwhile, the events become the system’s messages, so both

conversations have links to the classes *RWC* and *GetRWC*. Figure 60 depicts the resulting multi-agent Room Manager System.

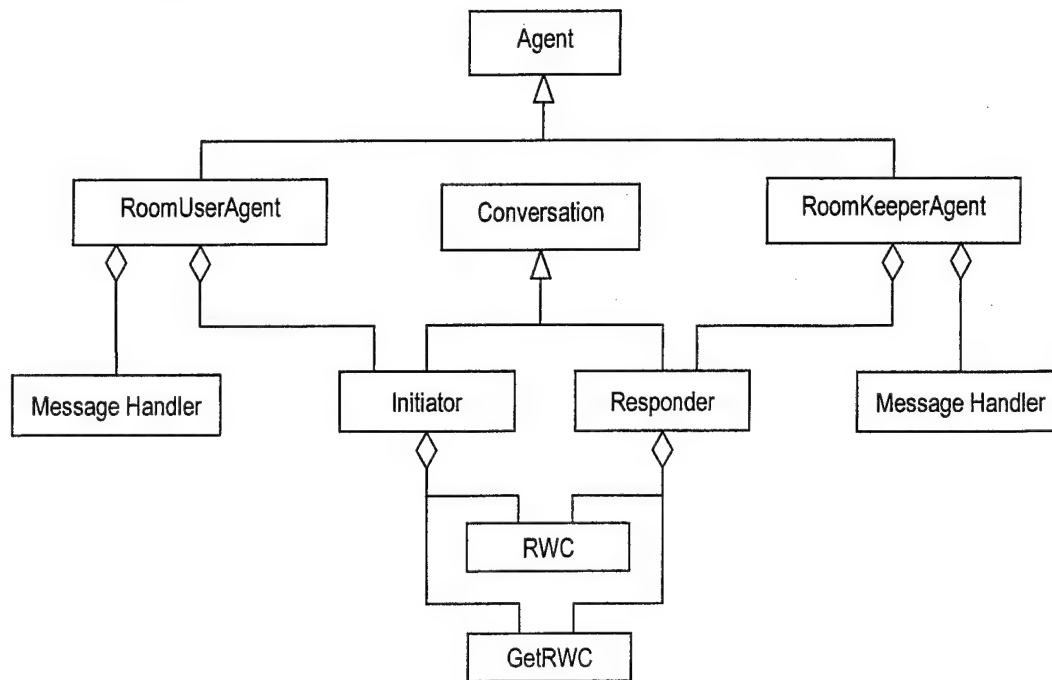


Figure 60. Multi-agent Room Manager System Implemented using agentMom

5.3.2. Room Manager and Channel Integration Demonstration. While the previous section shows a successful derivation of a multi-agent Room Manager system, it does not show the Room Manager model being integrated with the agentMom model. It is simply the creation of a system using the agentMom framework. As Section 5.3.1 points out, the problem is that one of the models does not follow the rules of a well-formed domain model. In order to overcome this drawback, it may be possible in some cases to modify existing models to make them fit into the required format. The following demonstration illustrates this approach.

Channel is a domain model that was developed as a “wrapper” to model the behavior evident in all/most multi-agent systems. The wrapper is designed to hide the details of the underlying agent frameworks. As long as the framework supports the provided interface, the Channel model can be integrated with other domain models. Figure 61 is the aggregate model

and Figure 62 is the model's message passing paradigm, and are provided to assist in the integration analysis with the Room Manager model. The Channel AWL domain model can be found in its entirety in Appendix E.

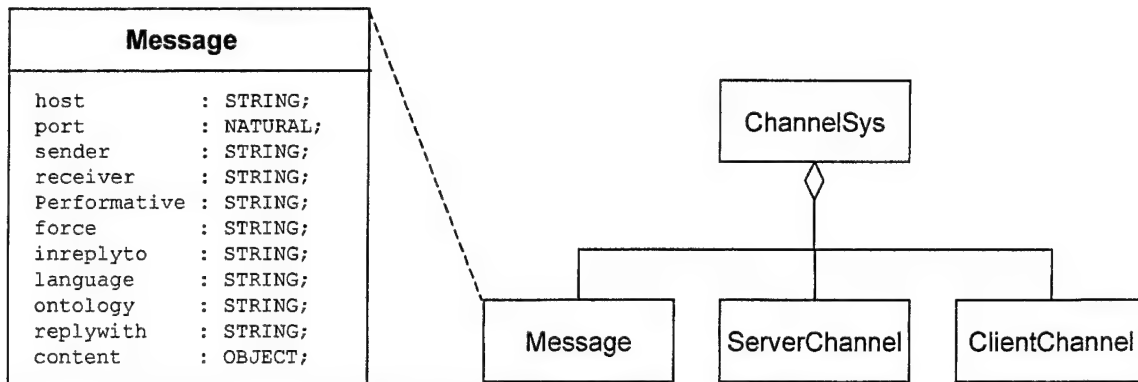


Figure 61. Channel Aggregate Model

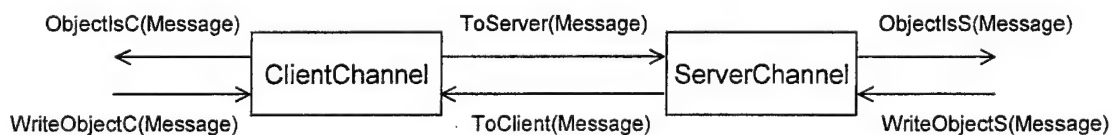


Figure 62. Channel Message Passing Diagram

The analysis results of Channel's interface contract showed that a combination of the *WriteObject(C/S)* and *ObjectIs(C/S)* events are necessary in order to integrate the models with the desired functionality. The *WriteObject* events are used to store message objects to the Channel and *ObjectIs* events are used to retrieve those messages for dissemination to the applicable agent, in this case RoomUser and RoomKeeper. Figure 63 is the result of the integration analysis between Room Manager and Channel. It shows four communication patterns, each a severed intra-model event connection. Room Manager's intra-model events: *GetRoom*, *GetRWC*, *NewRWC*, and *RWC* were identified as supplying/receiving information from Channel's *WriteObject* and *ObjectIs* events. Each interface fit the transfer communication pattern, and as an additional similarity, all pass a single message object.

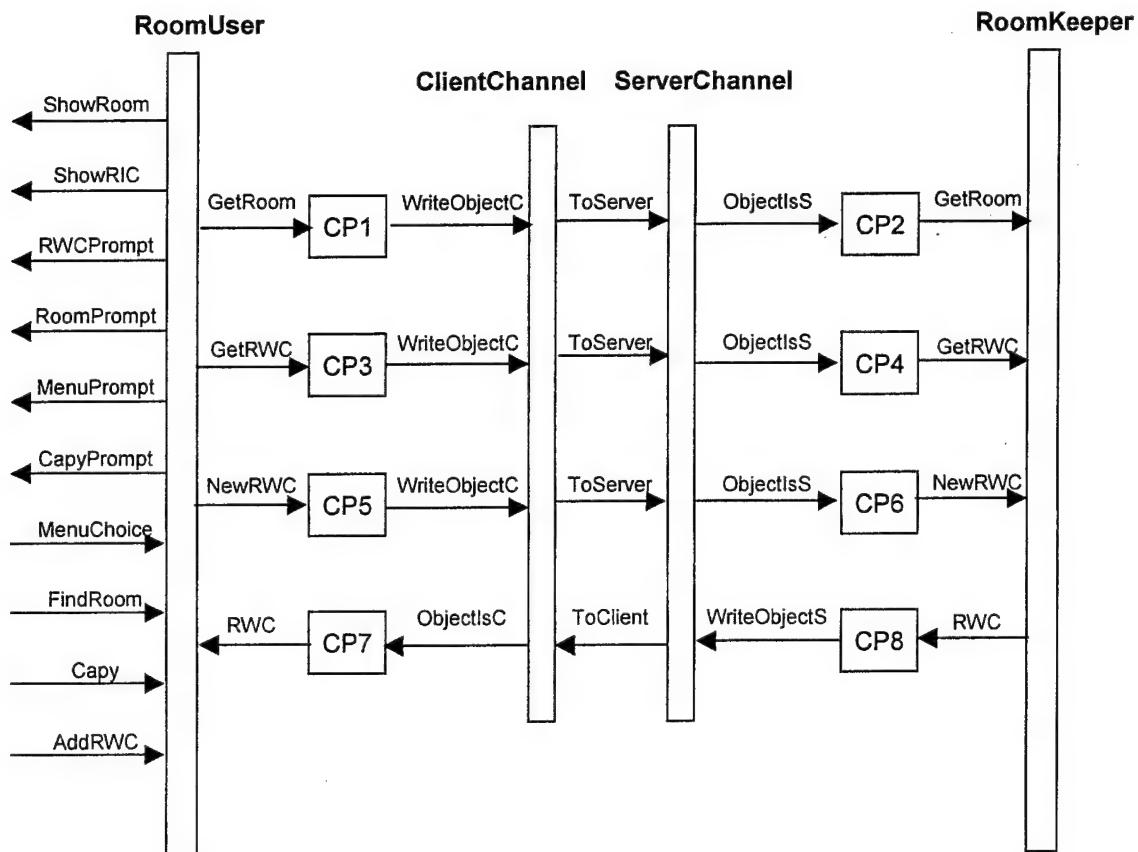


Figure 63. Room Manager / Channel Integration Analysis

Therefore, the complete integration requires eight separate, but similar, MICs. The differences occur due to the function of each MIC. There are two types of MICs, encoders and decoders. Each MIC's *ConvertParameterFunction* handles either the encoding or decoding of a message object to or from the Channel. The resulting MICs generated from the integration are shown following the Channel model in Appendix E. Figure 64 shows the screen shot taken from the ADMIT integration of these domain models.

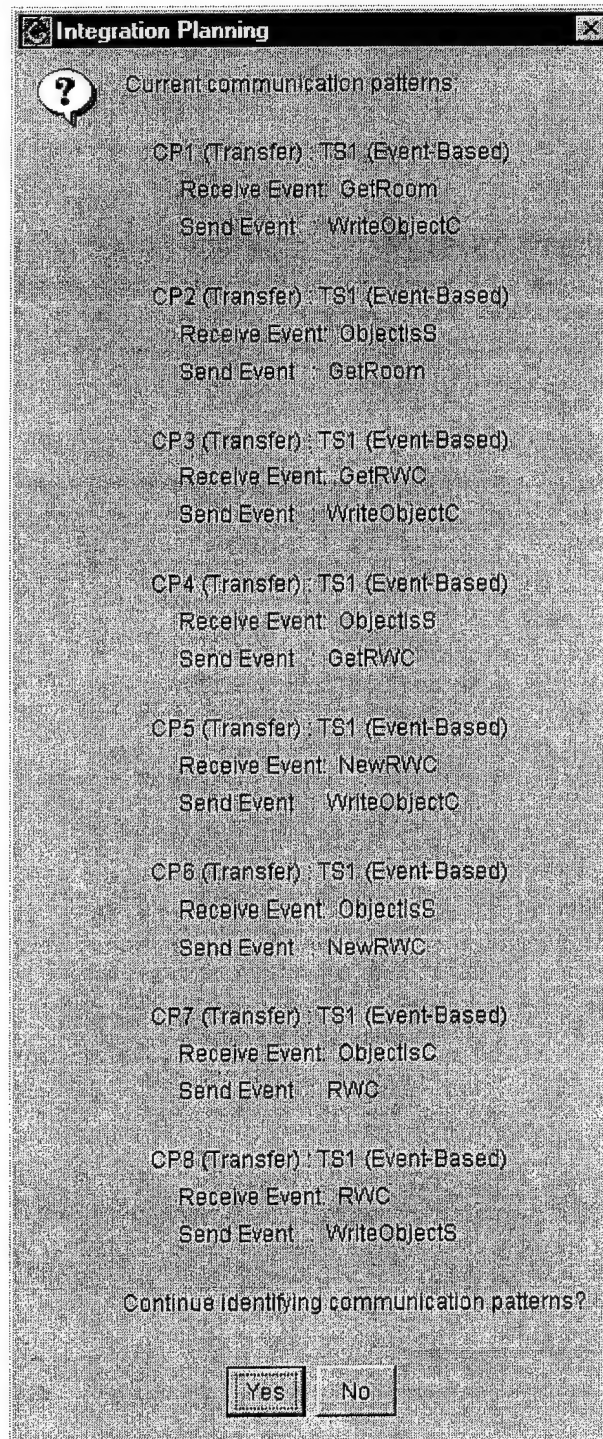


Figure 64. ADMIT Screen Capture (Integration of Room Manager and Channel)

6. Results, Conclusions, and Recommendations

6.1. Results

The goal of developing an integration methodology for domain models was successfully attained. Specifically, a generic domain model integration methodology was developed. The event based integration, that focused heavily on inter-model communication, lead to the development of a state-model based methodology. The generic, UML, methodology was then transformed into a AWL specific version. Finally the AWSOME Domain Model Integration Tool (ADMIT) was successfully created to demonstrate the feasibility of automating the methodology. Additionally, in order to create both the AWL integration methodology and ADMIT, rules for defining well-formed AWL domain models were created. From these rules, a support tool was developed to provide domain model verification. All of the models used as input for the demonstrations of this methodology were first checked using the verification software. Additionally, all of the resulting integrated domain models were also subjected to the model verifier to ensure that the models were not only syntactically correct, but that they also followed the domain model rules. Finally, the models created during the demonstrations were visually inspected to verify that they provided the desired functionality gained by the integration while the input models remained unchanged.

6.2. Conclusions

The use of this integration methodology can significantly aid application engineers in creating formal specifications by eliminating the need to develop specifications by-hand that appear in existing domain models. Additionally, when the specifications that satisfy the requirements in a given problem statement exist across multiple domains, the application engineers have at their disposal a tool that provides a great deal of flexibility in developing

application systems. Because maintenance that is performed on software early in the lifecycle is much cheaper, the integration of domain models at the analysis phase can provide great savings in terms of development time and resource allocation.

6.3. Synopsis

In order to successfully develop the methodology, the analysis of several key background topics were required. Those topics included: domain theory, software component theory, and type conversion theory.

Domain theory was needed to provide the definition of the domain model's structural, functional, and dynamic components for both the UML and AWL specified domains. Additionally, the AST representation of AWL models needed to be understood in order to verify the correctness of the AWL methodology as well as to create the demonstration integration tool, ADMIT. Included with the domain theory was a separate research effort to standardize the representation of domain models using AWL. The resulting eight rules, which defined well-formed domain models, were developed and used to ensure that the AWL version of the methodology was correct.

While an understanding of domain models was an integral part of developing the integration technique, other theories were required to assist in discovering how models could be combined. In order to address the issue of how to view domains, software component theory was used to supply the ideas of black-box components and component interfaces. A key concept in the problem statement was ensuring that the resulting integrated model was correct. In order to satisfy this requirement, the input models to the methodology were determined to be black-box components, which were then protected from modification, thus preserving their correctness. Additionally, the component interface contracts provided the ability for the methodology to determine both "where" to combine models as well as "what" conversions were required.

It was the need to convert the shared information between models that required the research of type theory. Specifically, the conversion of a value from one type to another requires that the range of the target type be adequate to handle all possible inputs from the source type. This is important because it governs how interface conversions must be designed in order to ensure their correctness. In addition to range restrictions, type bridging was defined to handle the cases where values could be converted directly into the target type, called type casting, as well as those cases where each input value must be individually dealt with, called type mapping.

Once those background topics were explored, it was determined that a new software entity was required that could convert the information being shared between models through their interfaces. Thus the model interface conversion or MIC concept was developed to deal with the issues of combining the input models' send and receive events. Specifically, the MIC was designed in such a way that the values contained in each of the MIC's receive events could be computed or converted to produce the information required for the MIC's send events.

6.4. Recommendations for Future Research

While the development of both the UML and AWL methodologies was successful, there are still many opportunities to expand this particular area of study. Further research is necessary to explore other possible integration techniques. Additionally, model integration can be used to help advance the study of automatic code generation systems. The following topics are just a few of the possible areas of future research concerning the integration of formal domain models.

6.4.1. Alternate Container Types (AWL Methodology). Section 4.3.6 describes how to create the MIC's attribute which is used to store the incoming values from the MIC's receive events. While the attribute type had to be a container, the choice as to which type was not clear. It could not be a set because values could be received by successive events where the values are

the same. A set would eliminate one of the values! The remaining choice was between a bag and a sequence. At the time, a bag seemed a better choice because of the lack of restrictions upon it. However, a problem was found in generating post-conditions where the order of values in the container was important. If a sequence were used, these expressions would be possible to generate.

6.4.2. Information Retrieval Techniques in Domain Analysis. Interface contracts are used to provide the application engineer with the knowledge of those events that are available to connect the input domain models. The drawback to this technique is the lack of standardization of such contracts. It is completely up to the domain expert to produce this documentation, and if produced, the contract format between one model and another may be so completely different that understanding them may impose an additional burden to the application engineer. Therefore, it would be preferable to have automated support tools that could interrogate models to extract their interfaces.

6.4.3. Artificial Intelligence Techniques in Domain Selection. Step one, the selection of input domain models in the integration methodology, assumes that the application engineer has some knowledge of a library of possible domain models from which to choose. While this may be the case, the size and thus the variety of models available may be limited. Also, as new domains are introduced, the application engineer may have to review each in order to have a complete picture of available domains. A better approach would be the use of agents that could independently navigate known domain repositories searching for possible input domain models. In this way, the responsibilities of the application engineer could be focused on accepting or rejecting proposed models. If this approach were combined with the domain model information retrieval techniques, the engineer could then be freed of all low-level decisions, with the exception of the communication pattern analysis between suggested input domain models.

6.4.4. *Verifying Correctness of Supplied Expressions.* This methodology could be enhanced to include correctness verification of the user supplied conversion expressions. In order for this approach to fully fit into formal methods, the user expressions must be subjected to correctness verification. Theorem proving techniques are available that should be able address this concern.

6.5. *Summary*

The end result of this research effort was the successful development of an integration methodology for formally specified domain models. The generic methodology applies to all UML based domain models, yet is extendable to apply to any specific domain modeling language. Further, the techniques applied to the integration process can readily be automated to provide software tools to assist application engineers in this effort.

Appendix A. AWSOME Wide Spectrum Language (AWL) Quick Reference

Types

Abstract

Syntax: type identifier is abstract;
Op: :=, =, /=

Enumeration

Syntax: type identifier is (identifier-1, ..., identifier-N);
Op: :=, =, /=, <, <=, >, >=
Notes: **Boolean** is predefined and has the following operations:
and, or, not, =>

Integer

Syntax: type identifier is range lower-bound .. upper-bound;
Op: :=, =, /=, <, <=, >, >=, +, -, *, /, **, mod, (unary -)
Notes: lower-bound, and upper-bound must be of some Integer
type (or * for no upper-bound)

Real

Syntax: type identifier is
(digits digits [base base] | delta delta)
range lower-bound .. upper-bound;
Op: :=, =, /=, <, <=, >, >=, +, -, *, /, **, (unary -)
Notes: lower-bound, and upper-bound must be of some Real
type, base and digits must be of some Integer type.

Container

Syntax: type identifier is
(bag | sequence | set) of element-type;
type identifier is array [index-type] of element-type;
Ops: ALL: :=, =, /=
array: element selection (a[i])
bag, sequence, set: membership (in, union, intersect)
Notes: index-type must be a Integer or Enumeration type

Access

Syntax: type identifier is access type-name;
Op: :=, =, /=, (dereferencing ptr^)
Notes: null is a predefined value for access types

Record

Syntax: type identifier is
Record
declaration-1;
..
declaration-N;
end record;
Ops: :=, =, /=, component selection (record.declaration)

Union

Syntax: type identifier is
 union
 I-Val : Integer;
 F-Val : Float;
 B-Val : Boolean;
 end union;
Ops: :=, =, /=, component selection

Declarations

Syntax: identifier : [constant] object-type [:= value]

Expressions

Literals constant

Integer: type = universal integer
Real: type = universal real
Character: type =
String: type =
Null: type = universal access

Binary (operands must be of same type (except universal types))

Numeric: (+, -, *, /, **, mod)
 Result is the most restrictive operand type
Comparison: (=, /=, <, <=, >, >=)
 Result is Boolean
Container: (In) 1st op can be any type, 2nd must be a bag
 Result is Boolean
 (subset, subseteq) ops must be same set type
 Result is Boolean
 (union, intersect) ops must be same set type
 Result is the same type as the operands
Logical: (and, or) ops must be Boolean
 Result is Boolean

Unary

Minus: (-) operand can be either integer or real
 Result is the same as the operand
Not: (not) operand must be a Boolean
 Result is Boolean

Quantified (used for pre- and post-conditions)

Syntax: [forall | exists | unique]
 ([logical-variables]*) (constraint)
Notes: constraint must be Boolean / Result is Boolean

Container Formers

Bag: { * term | ([logical-variable] *) (constraint) * }
Sequence: [term | ([logical-variable] *) (constraint)]
Set: { term | ([logical-variable] *) (constraint) }

Literal Containers

Bag: { * [term [, temp] *] * }
Sequence: [[term [, temp] *]]
Set: { [term [, temp] *] }

Function Call

Syntax: *function-name* ([argument [, argument] *]);
 arguments must match their respective parameter
 types. Result is the type returned by the function.

Allocator

Syntax: *new type-name*
 Result is an access type

Access

Syntax: *&object-name*
 Result is an access type

Type Conversion (see reference manual)

Subprograms

Procedure

Syntax: *procedure identifier* (*sequence of formal parameters*)
 [*assumes precondition*]
 [*guarantees postcondition*]
 [*is*
 [*local object declarations*]
 begin
 sequence of statements
 end;]

Function

Syntax: *function identifier* (*sequence of formal parameters*)
 : *return-type*
 [*assumes precondition*]
 [*guarantees postcondition*]
 [*is*
 [*local object declarations*]
 begin
 sequence of statements
 end;]

Statements

Assignment

Syntax: *name* := *expression*;

If-then-else

Syntax: if *condition* then
 sequence of statements
 [else
 sequence of statements]
 end if;

Loop

Syntax: while *condition* do
 sequence of statements
 end do;

Procedure Call

Syntax: *procedure-name* ([*argument* [, *argument*] *]);
 arguments must match their respective parameter
 types.

Goto

Syntax: goto *label*;

Units

Package

Syntax: package *identifier* is
 [*declaration* | package]*
 end package;

Class

Syntax: class *identifier* is [abstract] [superclass with]
 [*attribute* | *method*]*
 [invariant *condition*]
 [dynamic model]
 end class;

Attributes

Syntax: [public | private] *data-object-declaration*;

Methods

Syntax: (public | private) [abstract] [class]
 Subprogram-declaration;

Dynamic Model

Syntax: dynamic model is
 (event identifier (sequence of formal parameters)
 [assumes condition] ;)+
 (state identifier [invariant condition] ;)+
 transition table is
 (in from-state on event if guard do action
 [send (event) to to-state;]*
 end transition table;
 end dynamic model;

Association

Syntax: association identifier is
 (role identifier : [ordered] class-name
 multiplicity type-name;)2+
 [invariant condition]
 end association;

Aggregation

Syntax: aggregation identifier is
 parent identifier : [ordered] class-name
 multiplicity type-name;
 child identifier : [ordered] class-name
 multiplicity type-name;
 [invariant condition]
 end aggregation;

Associative Object

Syntax: assocobject identifier is
 (role identifier : [ordered] class-name
 multiplicity type-name
 [qualified by attribute] ;)2+
 (attribute | method)*
 [invariant condition]
 end assocobject;

Appendix B. AWSOME's WsClasses AST Structure

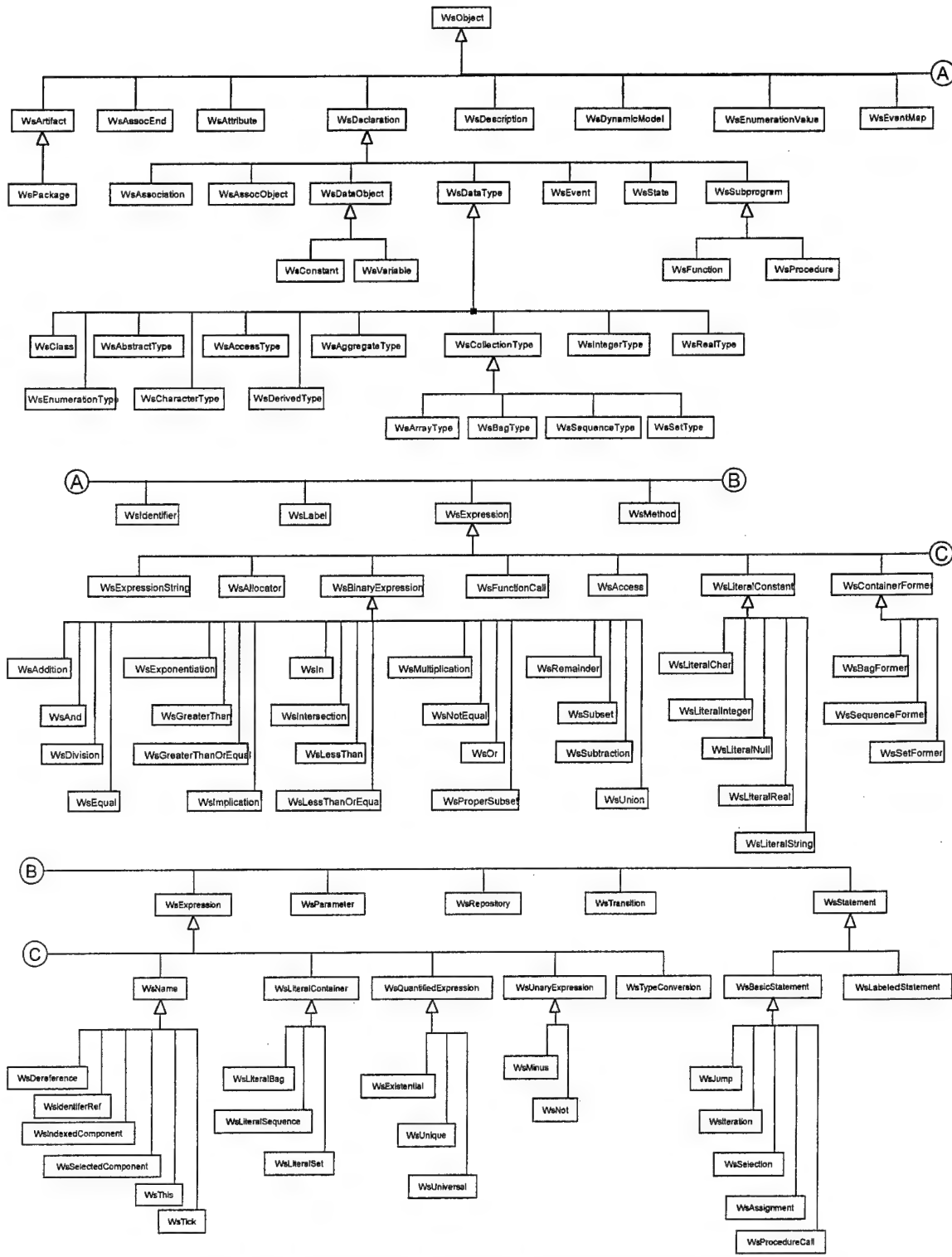


Figure 65. AWSOME's WsClasses AST Inheritance Diagram

Appendix C. Room Manager System Domain Model (AWL)

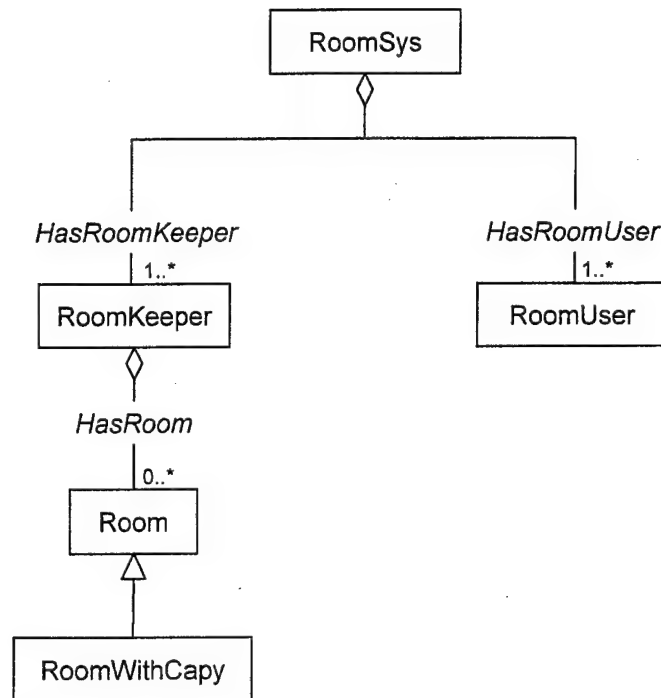


Figure 66. Room Manager System Aggregate Domain Model

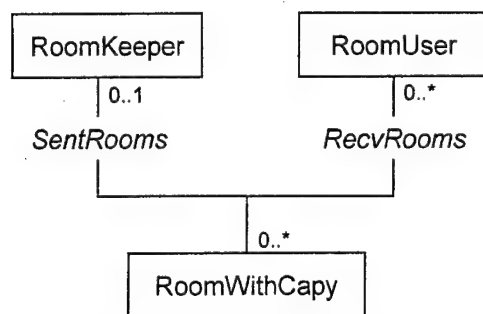


Figure 67. Room Manager System Association Model

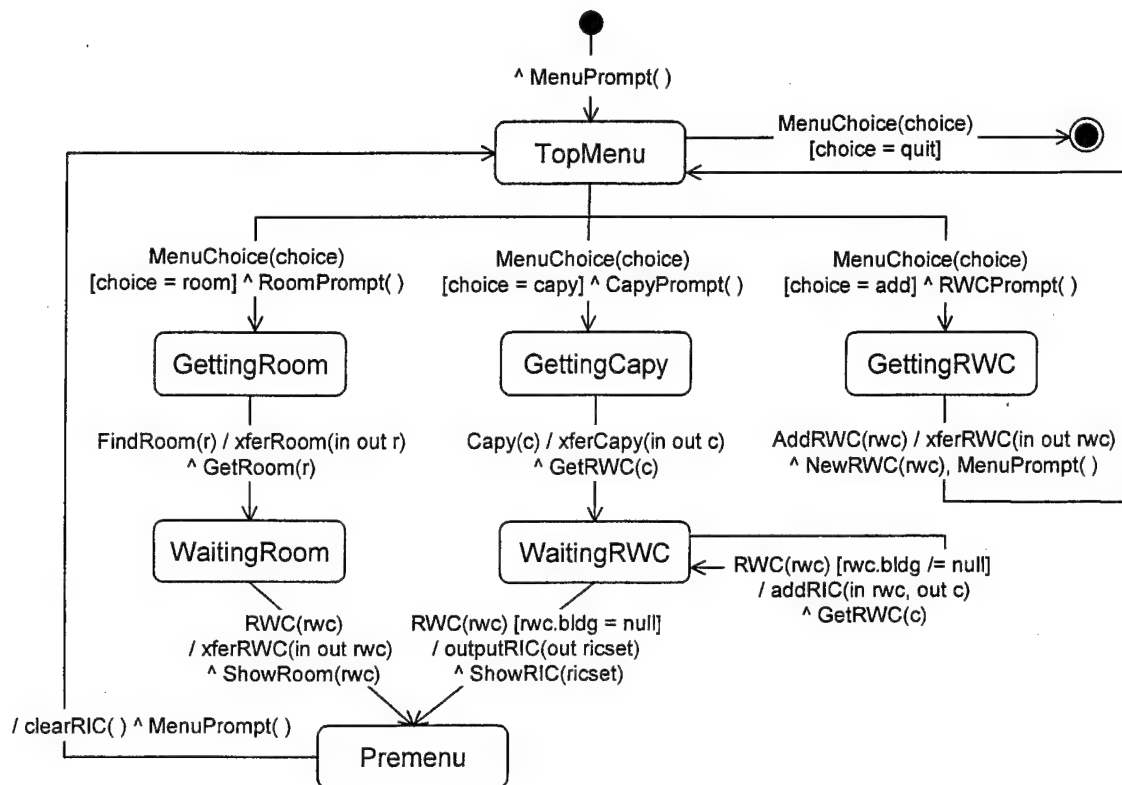


Figure 68. Room Manager's RoomUser State Diagram

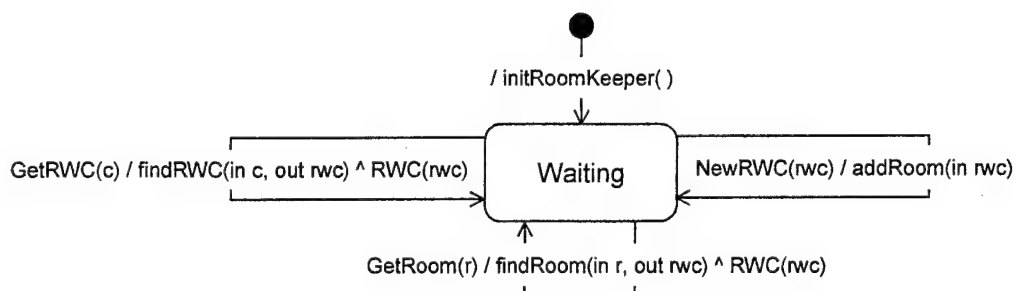


Figure 69. Room Manager's RoomKeeper State Diagram

/* THIS SOFTWARE AND ANY ACCOMPANYING DOCUMENTATION IS RELEASED "AS IS."
 * THE U.S. GOVERNMENT MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
 * CONCERNING THIS SOFTWARE AND ANY ACCOMPANYING DOCUMENTATION, INCLUDING,
 * WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A
 * PARTICULAR PURPOSE. IN NO EVENT WILL THE U.S. GOVERNMENT BE LIABLE
 * FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER
 * INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE, OR
 * INABILITY TO USE, THIS SOFTWARE OR ANY ACCOMPANYING DOCUMENTATION,
 * EVEN IF INFORMED IN ADVANCE OF THE POSSIBILITY OF SUCH DAMAGES.*/

/*****
 Room Manager System Domain Model (RoomSys.awl)

Description:

The Room Manager System maintains a set of rooms from which queries can be made. The Room has a name and a building number, and is extended into the RoomWithCapy, which adds the room's seating capacity. The User is allowed to query the room set for specific rooms or can view a returned set of rooms that satisfy a given capacity.

Interface Contract:

Event	Inter/Intra Parameters	Class Parameter Type	Description
AddRWC	Inter - RoomUser (receiver) rwc	RoomWithCapy	The answer to the prompt for a room to add.
Capy	Inter - RoomUser (receiver) c	NATURAL	The answer to the prompt for a capacity.
CapyPrompt	Inter - RoomUser (sender)		The prompt to the user for the capacity describing the rooms to show.
FindRoom	Inter - RoomUser (receiver) r	Room	The answer to the prompt for a room to find.
GetRoom	Intra r	Room	
GetRWC	Intra c	NATURAL	
MenuChoice	Inter - RoomUser (receiver) choice	MENUCHOICE	The answer to the prompt for a menu choice. Valid choices are: add, capy, room, and quit.
MenuPrompt	Inter - RoomUser (sender)		The menu prompt to the user.
NewRWC	Intra rwc	RoomWithCapy	
RoomPrompt	Inter - RoomUser (sender)		The prompt to the user for the room to show.
RWC	Intra rwc	RoomWithCapy	
RWCPrompt	Inter - RoomUser (sender)		The prompt to the user for the room to add.
ShowRIC	Inter - RoomUser (sender) ricset	RoomWithCapySetType	The output room set to the user.
ShowRoom	Inter - RoomUser (sender) rwc	RoomWithCapy	The output room to the user.

History:

Original - Created by Capt Marsh and maintained by Dr. Hartrum.
 12/11/00 - Updated to reflect the "well-formed" domain rules. The new system class "RoomSys" was added, and associations for each of the model's intra-model events were added. (Nonnweiler)
 12/19/00 - Added the Interface Contract to the model documentation (Nonnweiler)

*****/

```

package RoomMgrSystem is

  type CHARACTER is abstract;
  type STRING is sequence of CHARACTER;
  type MENUCHOICE is (add, capy, room, quit);
  type NATURAL is range 0 .. *;
  type Optional is range 0 .. 1;
  type ZeroOrMore is range 0 .. *;
  type One is range 1 .. 1;
  type RWCSetType is set of RoomWithCapy;

  function sizeOf(s : in RWCSetType) : NATURAL
  function cat(s1 : in STRING, s2 : in STRING) : STRING

  //-----
  class RoomSys is
    // This is the system class
  end class;

  //-----
  class Room is
    public bldg : STRING;
    public num : STRING;

    private procedure initRoom()
      guarantees bldg' = null and num' = null
    end class;

  //-----
  class RoomWithCapy is Room with
    public capacity : NATURAL;

    private procedure initRoomWithCapy()
      guarantees capacity' = 0
    end class;

  //-----
  class RoomKeeper is
    public size : NATURAL;

    private procedure initRoomKeeper()
      guarantees (this.HasRooms.roomSet' = { } and
        this.SentRooms.sentRoomSet' = { } and size' = 0)
    private procedure addRoom(rwc : in RoomWithCapy)
      guarantees rwc in this.HasRooms.roomSet'
    private procedure findRoom(r : in Room, rwc : out RoomWithCapy)
      guarantees exists (rm : RoomWithCapy)
        ((rm in this.HasRooms.roomSet and rm.bldg = r.bldg and
          rm.num = r.num and rwc = rm) or
          (not exists (rm : RoomWithCapy) (rm in this.HasRooms.roomSet and
            rm.bldg = r.bldg and rm.num = r.num and rwc.bldg = null
              and rwc.num = null))))
    private procedure findRWC(c : in NATURAL, rwc : out RoomWithCapy)
      guarantees exists (rm : RoomWithCapy)
        ((rm in this.HasRooms.roomSet and rm.bldg = rwc.bldg and rm.num = rwc.num and
          rm.capacity >= c and not rwc in this.SentRooms.sentRoomSet and
          rwc in this.SentRooms.sentRoomSet') or
          (not exists (rm : RoomWithCapy) (rm.capacity = c and
            not rm in this.SentRooms.sentRoomSet and rwc.bldg = null and
              rwc.num = null and this.SentRooms.sentRoomSet' = { }))))

    invariant (size = sizeOf(this.HasRooms.roomSet))

```

```

dynamic model is
    event AUTO();
    event NewRWC(rwc : in RoomWithCapy);
    event GetRoom(r : in Room);
    event GetRWC(c : in NATURAL);
    event RWC(rwc : out RoomWithCapy);

    state Waiting;
    state START;

    transition table is
        in START on AUTO do initRoomKeeper to Waiting;
        in Waiting on NewRWC do addRoom to Waiting;
        in Waiting on GetRoom do findRoom send RWC to Waiting;
        in Waiting on GetRWC do findRWC send RWC to Waiting;
    end transition table;
end dynamic model;
end class;

//-----
class RoomUser is
    public theCapy : NATURAL;

    private procedure initRoomUser()
        guarantees this.RecvRooms.roomsInConstraint' = { } and theCapy' = 0
    private procedure addRIC(rwc : in RoomWithCapy, c : out NATURAL)
        guarantees rwc in this.RecvRooms.roomsInConstraint' and c = theCapy
    private procedure clearRIC()
        guarantees this.RecvRooms.roomsInConstraint' = { }
    private procedure xferRoom(r : in out Room)
    private procedure xferRWC(rwc : in out RoomWithCapy)
    private procedure xferCapy(c : in out NATURAL)
        guarantees theCapy' = c
    private procedure outputRIC(ricset : out RoomWithCapySetType)
        guarantees ricset = this.RecvRooms.roomsInConstraint and
            this.RecvRooms.roomsInConstraint' = { }

dynamic model is
    event AUTO();
    event MenuPrompt();
    event MenuChoice(choice : in MENUCHOICE);
    event RWCPrompt();
    event CapyPrompt();
    event RoomPrompt();
    event FindRoom(r : in Room);
    event AddRWC(rwc : in RoomWithCapy);
    event NewRWC(rwc : out RoomWithCapy);
    event RWC(rwc : in RoomWithCapy);
    event Capy(c : in NATURAL);
    event GetRoom(r : out Room);
    event GetRWC(c : out NATURAL);
    event ShowRIC(ricset : out RoomWithCapySetType);
    event ShowRoom(rwc : out RoomWithCapy);

    state START;
    state END;
    state TopMenu;
    state GettingRWC;
    state GettingCapy;
    state GettingRoom;
    state WaitingRWC;
    state WaitingRoom;
    state Premenu;

```

```

transition table is
  in START on AUTO send MenuPrompt to TopMenu;
  in TopMenu on MenuChoice if (choice = add) send RWCPrompt to GettingRWC;
  in TopMenu on MenuChoice if (choice = capy) send CapyPrompt to GettingCapy;
  in TopMenu on MenuChoice if (choice = room) send RoomPrompt to GettingRoom;
  in TopMenu on MenuChoice if (choice = quit) to END;
  in GettingRWC on AddRWC do xferRWC send NewRWC, MenuPrompt to TopMenu;
  in GettingCapy on Capy do xferCapy send GetRWC to WaitingRWC;
  in GettingRoom on FindRoom do xferRoom send GetRoom to WaitingRoom;
  in WaitingRWC on RWC if (rwc.bldg /= null) do addRIC send GetRWC to
    WaitingRWC;
  in WaitingRWC on RWC if (rwc.bldg = null) do outputRIC send ShowRIC to
    Premenu;
  in WaitingRoom on RWC do xferRWC send ShowRoom to Premenu;
  in Premenu on AUTO do clearRIC send MenuPrompt to TopMenu;
end transition table;
end dynamic model;
end class;

//-----
aggregation HasRooms is
  parent keeper: RoomKeeper multiplicity Optional;
  child roomSet: RoomWithCapy multiplicity ZeroOrMore;
end aggregation;

aggregation HasRoomUser is
  parent p : RoomSys multiplicity One;
  child c : RoomUser multiplicity OneOrMore;
end aggregation;

aggregation HasRoomKeeper is
  parent p : RoomSys multiplicity One;
  child c : RoomKeeper multiplicity OneOrMore;
end aggregation;

//-----
association SentRooms is
  role keeper: RoomKeeper multiplicity Optional;
  role sentRoomSet: RoomWithCapy multiplicity ZeroOrMore;
end association;

association RecvRooms is
  role keeper: RoomUser multiplicity ZeroOrMore;
  role roomsInConstraint: RoomWithCapy multiplicity ZeroOrMore;
end association;

association GetRWC is
  role s1 : RoomUser multiplicity One;
  role r1 : RoomKeeper multiplicity One;
end association;

association RWC is
  role s1 : RoomKeeper multiplicity One;
  role r1 : RoomUser multiplicity One;
end association;

association GetRoom is
  role s1 : RoomUser multiplicity One;
  role r1 : RoomKeeper multiplicity One;
end association;

association NewRWC is
  role s1 : RoomUser multiplicity One;
  role r1 : RoomKeeper multiplicity One;
end association;

end package;

```

Appendix D. Security Manager System Domain Model (AWL)

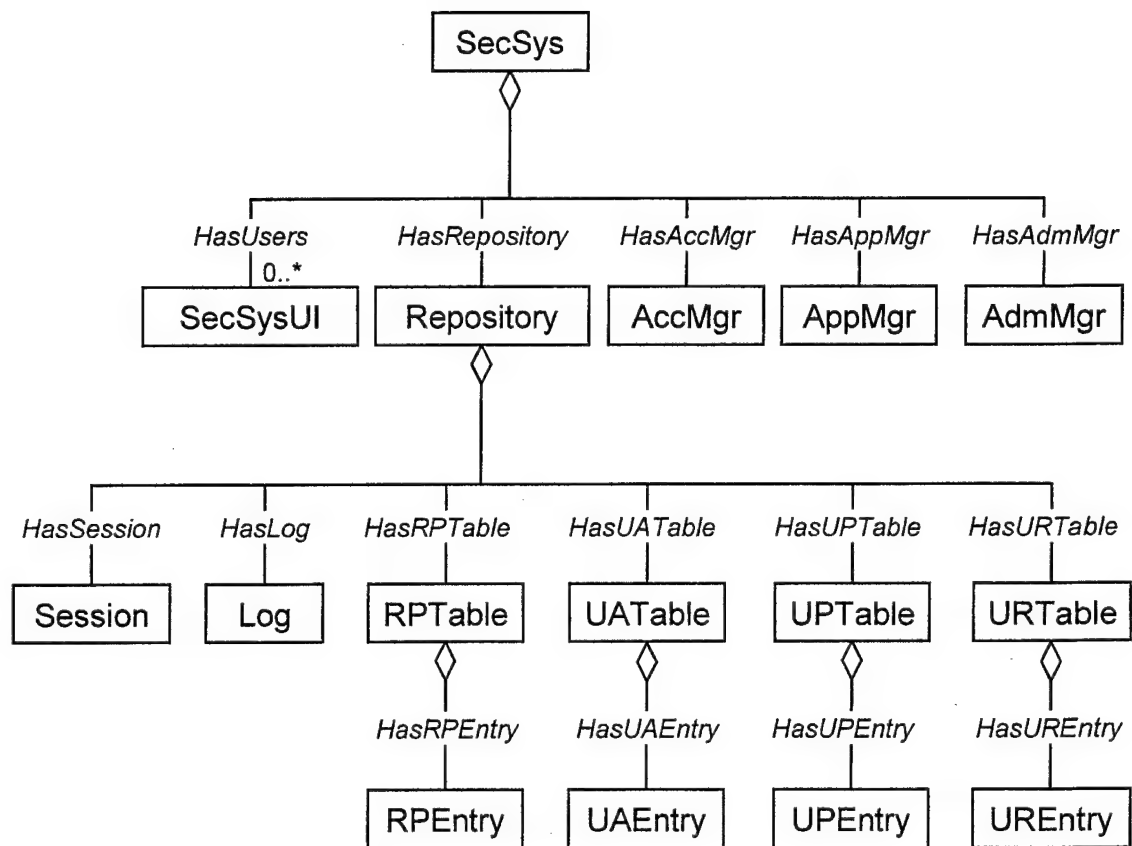


Figure 70. Security Manager's Aggregate Domain Model

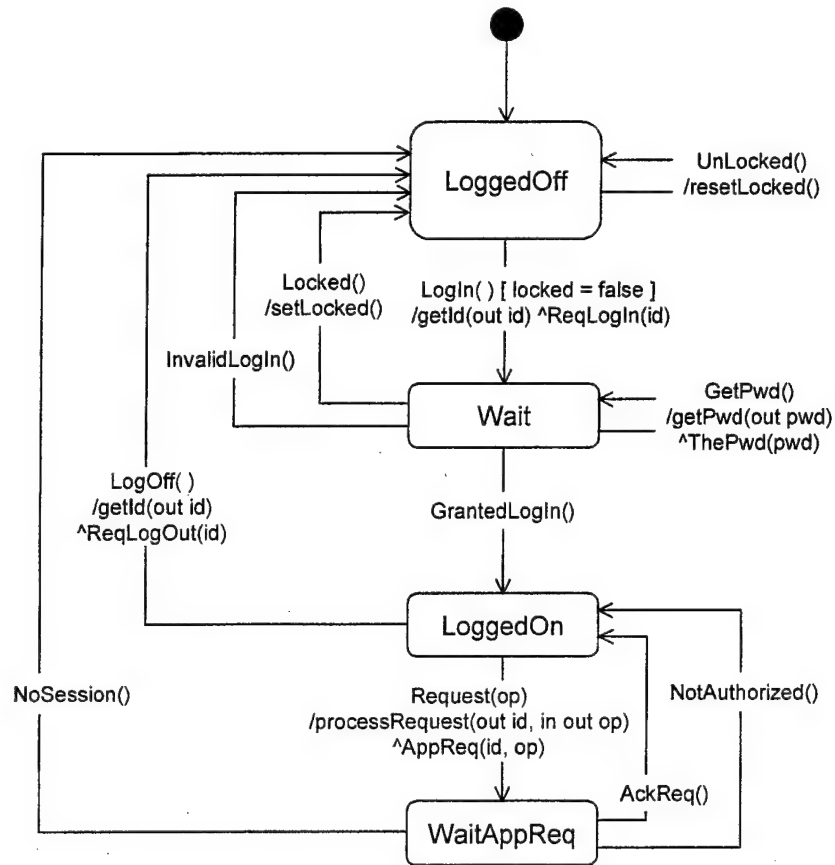


Figure 71. Security Manager's SecSysUI State Diagram

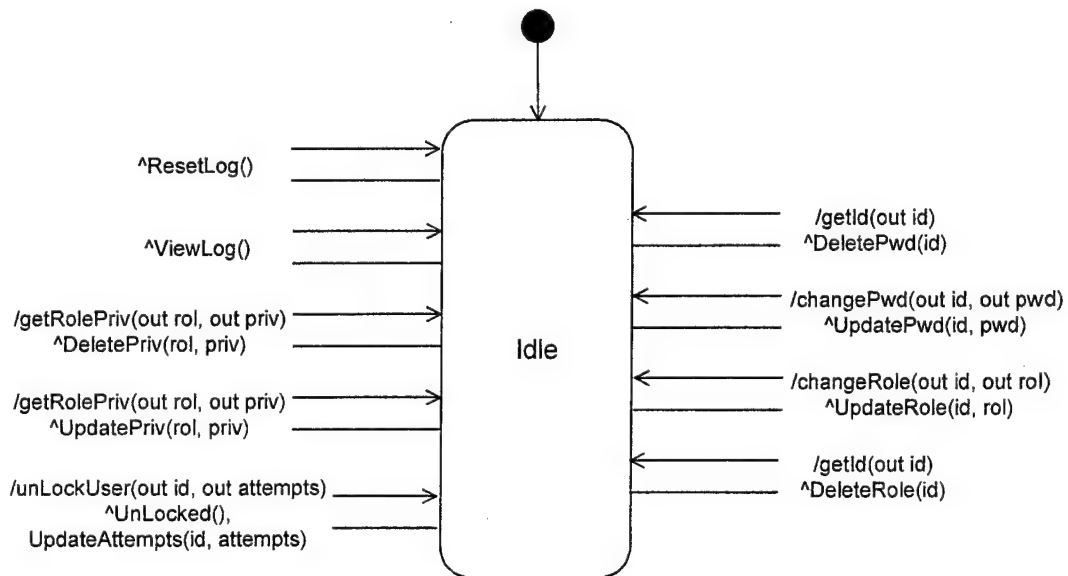


Figure 72. Security Manager's AdmMgr State Diagram

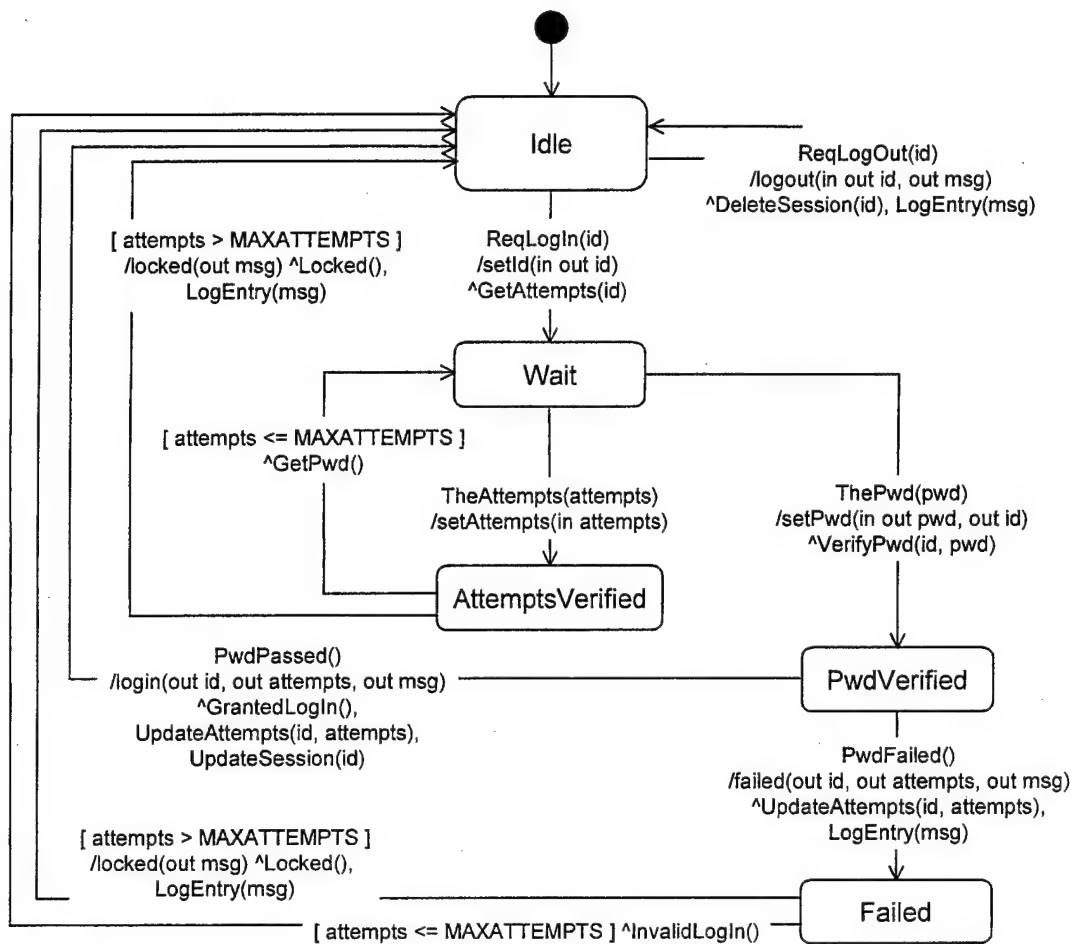


Figure 73. Security Manager's AccMgr State Diagram

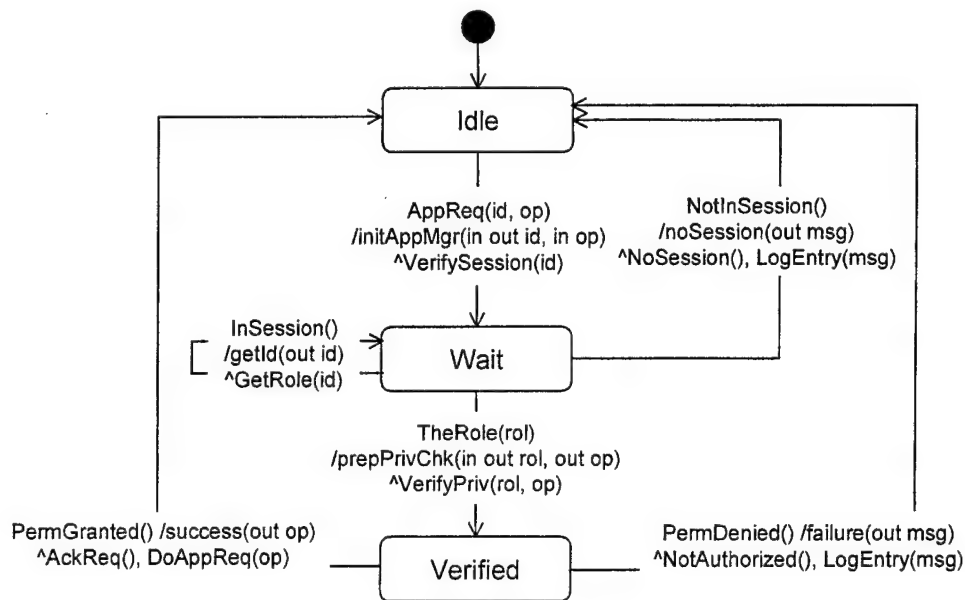


Figure 74. Security Manager's AppMgr State Diagram

/* THIS SOFTWARE AND ANY ACCOMPANYING DOCUMENTATION IS RELEASED "AS IS."
 * THE U.S. GOVERNMENT MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
 * CONCERNING THIS SOFTWARE AND ANY ACCOMPANYING DOCUMENTATION, INCLUDING,
 * WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A
 * PARTICULAR PURPOSE. IN NO EVENT WILL THE U.S. GOVERNMENT BE LIABLE
 * FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER
 * INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE, OR
 * INABILITY TO USE, THIS SOFTWARE OR ANY ACCOMPANYING DOCUMENTATION,
 * EVEN IF INFORMED IN ADVANCE OF THE POSSIBILITY OF SUCH DAMAGES.*/

/******
 Security System Domain Model (SecSys.awl)

Description:

The Security Manager System is designed to be integrated with other application domain models. It was created to demonstrate the ability to introduce a security protocol to applications that were created with no inherent security. In addition to this goal, an additional purpose for this model was to have a suitable domain model for integration with other models that were created independent of this model.

It supplies security in two fashions. In the first, users are required to "log-in" prior to being granted access to the protected application. The second verifies the user's privilege before each application request to ensure the proper authorization. As a necessary as side, the system must also allow for the administrator to unlock users, and to change the user profiles.

In order to implement the above goals, the Security System has three managers: AccessManager (AccMgr), ApplicationManager (AppMgr), and AdministrationManager (AdmMgr). Additionally, the system requires the use of a repository that contains the following tables: Session, Log, UserPasswords, UserRoles, UserAttempts, and RolePrivileges. The tables allow the various managers to view and update the current situation of any user.

Interface Contract:

Event	Intra / Inter - Class (s / r)		Description
	Parameters	Parameter Type	
AckReq	Intra		
AppReq	Intra		
	id	STRING	
	op	STRING	
DeletePriv	Intra		
	rol	STRING	
	priv	STRING	
DeletePwd	Intra		
	id	STRING	
DeleteRole	Intra		
	id	STRING	
DeleteSession	Intra		
	id	STRING	
DoAppReq	Inter - AppMgr (sender)		Sends the authorized operation to the protected application.
	op	STRING	
GetAttempts	Intra		
	id	STRING	
GetPwd	Intra		
GetRole	Intra		
	id	STRING	
GrantedLogIn	Intra		
InSession	Intra		
InvalidLogIn	Intra		
Locked	Intra		
LogEntry	Intra		
	msg	STRING	
LogIn	Inter - SecSysUI (receiver)		Indicates that the user wants to log on to the protected application.

LogOff	Inter - SecSysUI (receiver)		Indicates that the user wants to log off from the protected application.
NoSession	Intra		
NotAuthorized	Intra		
NotInSession	Intra		
PermDenied	Intra		
PermGranted	Intra		
PwdFailed	Intra		
PwdPassed	Intra		
ReqLogIn	Intra		
	id	STRING	
ReqLogOut	Intra		
	id	STRING	
Request	Inter - SecSysUI (receiver)		The users is requesting that operation op is to be performed by the protected application.
	op	STRING	
ResetLog	Intra		
TheAttempts	Intra		
	attempts	NATURAL	
ThePwd	Intra		
	pwd	STRING	
TheRole	Intra		
	rol	STRING	
Unlocked	Intra		
UpdateAttempts	Intra		
	id	STRING	
	attempts	NATURAL	
UpdatePriv	Intra		
	rol	STRING	
	priv	STRING	
UpdatePwd	Intra		
	id	STRING	
	pwd	STRING	
UpdateRole	Intra		
	id	STRING	
	rol	STRING	
UpdateSession	Intra		
	id	STRING	
VerifyPriv	Intra		
	rol	STRING	
	op	STRING	
VerifyPwd	Intra		
	id	STRING	
	pwd	STRING	
VerifySession	Intra		
	id	STRING	
ViewLog	Intra		

History:

06/30/00 - Original (Nonnweiler)
12/11/00 - New baseline (Nonnweiler)

*****/

package SecuritySystem is

```

type CHARACTER is abstract;
type STRING is sequence of CHARACTER;
type IdSetType is set of STRING;
type LogSetType is set of STRING;
type NATURAL is range 0 .. *;
type One is range 1 .. 1;
type ZeroOrMore is range 0 .. *;
```

```

MAX      : constant NATURAL := 3;
LOGOUTMSG : constant STRING := "LOGGED OUT";
LOCKEDMSG : constant STRING := "LOCKED";
LOGINMSG  : constant STRING := "LOGGED IN";
INVALIDMSG : constant STRING := "INVALID LOGIN ATTEMPT";
NOSESSMSG : constant STRING := "NO SESSION";
FAILMSG   : constant STRING := "NOT AUTHORIZED FOR: ";

function cat(s1 : in STRING, s2 : in STRING) : STRING
function inc(n : in NATURAL) : NATURAL

//-----
class SecSys is abstract
  // This is the system class.
end class;

//-----
class Repository is abstract
end class;

//-----
class SecSysUI is
  private id : STRING;
  private locked : BOOLEAN;

  private procedure setLocked()
    guarantees (this.locked' = true)
  private procedure resetLocked()
    guarantees (this.locked' = false)
  private procedure processRequest(id : out STRING, op : in out STRING)
    guarantees (id' = this.id)
  private procedure getId(id : out STRING)
    guarantees (id' = this.id)
  private procedure getPwd(pwd : out STRING)
    guarantees (pwd' /= NULL)

  dynamic model is
    event Auto();
    event Locked();
    event LogIn();
    event LogOff();
    event InvalidLogIn();
    event UnLocked();
    event ReqLogIn(id : out STRING);
    event GetPwd();
    event ThePwd(pwd : out STRING);
    event GrantedLogIn();
    event NotAuthorized();
    event AckReq();
    event Request(op : in STRING);
    event ReqLogOut(id : out STRING);
    event AppReq(id : out STRING, op : out STRING);
    event NoSession();

    state Start;
    state LoggedOff;
    state Wait;
    state LoggedOn;
    state WaitAppReq;

    transition table is
      in Start on Auto to LoggedOff;
      in LoggedOff on LogIn if locked = False do getId send ReqLogIn to Wait;
      in LoggedOff on UnLocked do resetLocked to LoggedOff;
      in Wait on Locked do setLocked to LoggedOff;
      in Wait on InvalidLogIn to LoggedOff;
      in Wait on GetPwd do getId send ThePwd to Wait;
      in Wait on GrantedLogIn to LoggedOn;
      in LoggedOn on Request do processRequest send AppReq to WaitAppReq;
      in LoggedOn on LogOff do getId send ReqLogOut to LoggedOff;
      in WaitAppReq on AckReq to LoggedOn;

```

```

        in WaitAppReq on NotAuthorized to LoggedOn;
        in WaitAppReq on NoSession to LoggedOff;
    end transition table;
end dynamic model;
end class;

//-----
class AccMgr is
    private id : STRING;
    private pwd : STRING;
    private attempts : NATURAL;

    private procedure logout(id : in out STRING, msg : out STRING)
        guarantees (this.id' = id and msg' = cat(id, LOGOUTMSG))
    private procedure setId(id : in out STRING)
        guarantees (this.id' = id)
    private procedure setAttempts(attempts : in NATURAL)
        guarantees (this.attempts' = attempts)
    private procedure setPwd(pwd : in out STRING, id : out STRING)
        guarantees (this.pwd' = pwd and id' = this.id)
    private procedure locked(msg : out STRING)
        guarantees (msg' = cat(this.id, LOCKEDMSG))
    private procedure login(id : out STRING, attempts : out NATURAL, msg : out STRING)
        guarantees (id' = this.id and attempts' = 0 and msg' = cat(this.id, LOGINMSG))
    private procedure failed(id : out STRING, attempts : out NATURAL, msg : out STRING)
        guarantees (id' = this.id and attempts' = inc(this.attempts) and
            msg' = cat(this.id, INVALIDMSG))

dynamic model is
    event Auto();
    event DeleteSession(id : out STRING);
    event GetAttempts(id : out STRING);
    event GetPwd();
    event GrantedLogIn();
    event InvalidLogIn();
    event Locked();
    event LogEntry(msg : out STRING);
    event PwdFailed();
    event PwdPassed();
    event ReqLogIn(id : in STRING);
    event ReqLogOut(id : in STRING);
    event TheAttempts(attempts : in NATURAL);
    event ThePwd(pwd : in STRING);
    event UpdateAttempts(id : out STRING, attempts : out NATURAL);
    event UpdateSession(id : out STRING);
    event VerifyPwd(id : out STRING, pwd : out STRING);

    00state Start;
    state Idle;
    state Wait;
    state AttemptsVerified;
    state PwdVerified;
    state Failed;

    transition table is
        in Start on Auto to Idle;
        in Idle on ReqLogOut do logout send DeleteSession, LogEntry to Idle;
        in Idle on ReqLogIn do setId send GetAttempts to Wait;
        in Wait on TheAttempts do setAttempts to AttemptsVerified;
        in AttemptsVerified on Auto if attempts > MAX do locked send Locked, LogEntry
            to Idle;
        in AttemptsVerified on Auto if attempts <= MAX send GetPwd to Wait;
        in Wait on ThePwd do setPwd send VerifyPwd to PwdVerified;
        in PwdVerified on PwdPassed do login send GrantedLogIn, UpdateAttempts,
            UpdateSession, LogEntry to Idle;
        in PwdVerified on PwdFailed do failed send UpdateAttempts, LogEntry to Failed;
        in Failed on Auto if attempts > MAX do locked send Locked, LogEntry to Idle;
        in Failed on Auto if attempts <= MAX send InvalidLogIn to Idle;
    end transition table;
end dynamic model;
end class;

```

```

//-----
class AppMgr is
  private id : STRING;
  private op : STRING;

  private procedure initAppMgr(id : in out STRING, op : in STRING)
    guarantees (this.id' = id and this.op' = op)
  private procedure noSession(msg : out STRING)
    guarantees (msg' = cat(id, NOSESSMSG))
  private procedure getId(id : out STRING)
    guarantees (id' = this.id)
  private procedure prepPrivChk(rol : in out STRING, op : out STRING)
    guarantees (op' = this.op)
  private procedure success(op : out STRING)
    guarantees (op' = this.op)
  private procedure failure(msg : out STRING)
    guarantees (msg' = cat(this.id, cat(FAILMSG, this.op)))

  dynamic model is
    event AckReq();
    event AppReq(id : in STRING, op : in STRING);
    event Auto();
    event DoAppReq(op : out STRING);
    event GetRole(id : out STRING);
    event InSession();
    event LogEntry(msg : out STRING);
    event NoSession();
    event NotAuthorized();
    event NotInSession();
    event PermDenied();
    event PermGranted();
    event TheRole(rol : in STRING);
    event VerifyPriv(rol : out STRING, op : out STRING);
    event VerifySession(id : out STRING);

    state Start;
    state Idle;
    state Wait;
    state Verified;

    transition table is
      in Start on Auto to Idle;
      in Idle on AppReq do initAppMgr send VerifySession to Wait;
      in Wait on NotInSession do noSession send NoSession, LogEntry to Idle;
      in Wait on InSession do getId send GetRole to Wait;
      in Wait on TheRole do prepPrivChk send VerifyPriv to Verified;
      in Verified on PermGranted do success send AckReq, DoAppReq to Idle;
      in Verified on PermDenied do failure send NotAuthorized, LogEntry to Idle;
    end transition table;
  end dynamic model;
end class;

```

```

//-----
class AdmMgr is
  private procedure getId(id : out STRING)
    guarantees (id' /= NULL)
  private procedure changePwd(id : out STRING, pwd : out STRING)
    guarantees (id' /= NULL and pwd' /= NULL)
  private procedure changeRole(id : out STRING, rol : out STRING)
    guarantees (id' /= NULL and rol' /= NULL)
  private procedure unlockUser(id : out STRING, attempts : out NATURAL)
    guarantees (id' /= NULL and attempts' = 0)
  private procedure getRolePriv(rol : out STRING, priv : out STRING)
    guarantees (rol' /= NULL and priv' /= NULL)

  dynamic model is
    event Auto();
    event DeletePriv(rol : out STRING, priv : out STRING);
    event DeletePwd(id : out STRING);
    event DeleteRole(id : out STRING);

```

```

    event ResetLog();
    event Unlocked();
    event UpdateAttempts(id : out STRING, attempts : out NATURAL);
    event UpdatePriv(rol : out STRING, priv : out STRING);
    event UpdatePwd(id : out STRING, pwd : out STRING);
    event UpdateRole(id : out STRING, rol : out STRING);
    event ViewLog();

    state Start;
    state Idle;

    transition table is
        in Start on Auto to Idle;
        in Idle on Auto do getId send DeletePwd to Idle;
        in Idle on Auto do changePwd send UpdatePwd to Idle;
        in Idle on Auto do getId send DeleteRole to Idle;
        in Idle on Auto do changeRole send UpdateRole to Idle;
        in Idle on Auto do getRolePriv send DeletePriv to Idle;
        in Idle on Auto do getRolePriv send UpdatePriv to Idle;
        in Idle on Auto send ResetLog to Idle;
        in Idle on Auto send ViewLog to Idle;
        in Idle on Auto do unlockUser send Unlocked, UpdateAttempts to Idle;
    end transition table;
end dynamic model;
end class;

//-----
class Log is
    private log : LogSetType;

    private procedure viewLog()
    private procedure resetLog()
        guarantees (this.log' = {})
    private procedure addEntry(msg : in STRING)
        guarantees ({msg} in this.log')

    dynamic model is
        event Auto();
        event LogEntry(msg : in STRING);
        event ResetLog();
        event ViewLog();

        state Start;
        state Idle;

        transition table is
            in Start on Auto to Idle;
            in Idle on ViewLog do viewLog to Idle;
            in Idle on ResetLog do resetLog to Idle;
            in Idle on LogEntry do addEntry to Idle;
        end transition table;
    end dynamic model;
end class;

//-----
class Session is
    private found : BOOLEAN;
    private idSet : IdSetType;

    private procedure deleteSession(id : in STRING)
        guarantees (id intersect this.idSet' = {})
    private procedure addSession(id : in STRING)
        guarantees (id in this.idSet')
    private procedure verifySession(id : in STRING)
        guarantees ((this.found' = true and id in this.idSet) or
            (this.found' = false and id intersect this.idSet = {}))

    dynamic model is
        event Auto();
        event DeleteSession(id : in STRING);
        event InSession();

```

```

event NoSession();
event UpdateSession(id : in STRING);
event VerifySession(id : in STRING);

state Start;
state Idle;
state Verified;

transition table is
  in Start on Auto to Idle;
  in Idle on DeleteSession do deleteSession to Idle;
  in Idle on UpdateSession do addSession to Idle;
  in Idle on VerifySession do verifySession to Verified;
  in Verified on Auto if found = True send InSession to Idle;
  in Verified on Auto if found = False send NoSession to Idle;
end transition table;
end dynamic model;
end class;

//-----
class UEntry is
  private id : STRING;
  private attempts : NATURAL;
end class;

//-----
class UTable is
  private procedure getAttempts(id : in STRING, attempts : out NATURAL)
    guarantees exists (e : UEntry)
      ((e.id = id and e in this.HasUEntry.c and attempts = e.attempts) or
       (e.id = id and e intersect this.HasUEntry.c = {} and attempts = 0))
  private procedure updateAttempts(id : in STRING, attempts : in NATURAL)
    guarantees exists (e : UEntry)
      (e.id = id and
       (attempts = 0 and e intersect this.HasUEntry.c = {}) or
       (attempts /= 0 and e.attempts' = attempts and e in this.HasUEntry.c))

dynamic model is
  event Auto();
  event GetAttempts(id : in STRING);
  event TheAttempts(attempts : out NATURAL);
  event UpdateAttempts(id : in STRING, attempts : in NATURAL);

  state Start;
  state Idle;

  transition table is
    in Start on Auto to Idle;
    in Idle on GetAttempts do getAttempts send TheAttempts to Idle;
    in Idle on UpdateAttempts do updateAttempts to Idle;
  end transition table;
end dynamic model;
end class;

//-----
class UEntry is
  private id : STRING;
  private pwd : STRING;
end class;

//-----
class UTable is
  private matched : BOOLEAN;

  private procedure updatePwd(id : in STRING, pwd : in STRING)
    guarantees exists (e : UEntry)
      (e.id = id and e.pwd' = pwd and e in this.HasUEntry.c)
  private procedure deletePwd(id : in STRING)
    guarantees forall (e : UEntry)
      (e.id = id and e intersect this.HasUEntry.c = {})
  private procedure verifyPwd(id : in STRING, pwd : in STRING)

```



```

    assumes exists (e : UPEEntry)
      (e.id = id and e in this.HasUPEEntry.c)
    guarantees exists (e : UPEEntry)
      ((e.id = id and e.pwd = pwd and this.matched = true) or
       (e.id = id and e.pwd /= pwd and this.matched = false))

dynamic model is
  event Auto();
  event DeletePwd(id : in STRING);
  event PwdFailed();
  event PwdPassed();
  event UpdatePwd(id : in STRING, pwd : in STRING);
  event VerifyPwd(id : in STRING, pwd : in STRING);

  state Start;
  state Idle;
  state Verified;

  transition table is
    in Start on Auto to Idle;
    in Idle on UpdatePwd do updatePwd to Idle;
    in Idle on DeletePwd do deletePwd to Idle;
    in Idle on VerifyPwd do verifyPwd to Verified;
    in Verified on Auto if matched = True send PwdPassed to Idle;
    in Verified on Auto if matched = False send PwdFailed to Idle;
  end transition table;
end dynamic model;
end class;

//-----
class UREEntry is
  private id : STRING;
  private rol : STRING;
end class;

//-----
class URTTable is
  private procedure deleteRole(entry : in UREEntry)
    guarantees exists (e : UREEntry)
      (e.id = entry.id and e.rol = entry.rol and e intersect this.HasUREEntry.c = {})
  private procedure updateRole(entry : in UREEntry)
    guarantees exists (e : UREEntry)
      (e.id = entry.id and e.rol = entry.rol and e in this.HasUREEntry.c)
  private procedure getRole(entry : in UREEntry, rol : out STRING)
    assumes exists (e : UREEntry)
      (e.id = entry.id and e.rol = entry.rol and e in this.HasUREEntry.c)
    guarantees exists (e : UREEntry)
      (e.id = entry.id and e.rol = entry.rol and
       e in this.HasUREEntry.c and rol = e.rol)

dynamic model is
  event Auto();
  event DeleteRole(entry : in UREEntry);
  event GetRole(entry : in UREEntry);
  event TheRole(rol : out STRING);
  event UpdateRole(entry : in UREEntry);

  state Start;
  state Idle;

  transition table is
    in Start on Auto to Idle;
    in Idle on UpdateRole do updateRole to Idle;
    in Idle on DeleteRole do deleteRole to Idle;
    in Idle on GetRole do getRole send TheRole to Idle;
  end transition table;
end dynamic model;
end class;

```

```

//-----
class RPEntry is
  private rol : STRING;
  private priv : STRING;
end class;

//-----
class RPTable is
  private authorized : BOOLEAN;

  private procedure deletePriv(entry : in RPEntry)
    guarantees (entry intersect this.HasRPEntry.c = {})
  private procedure updatePriv(entry : in RPEntry)
    guarantees exists (e : RPEntry)
      (e in this.HasRPEntry.c and e.rol' = entry.rol and e.priv' = entry.priv)
  private procedure verifyPriv(entry : in RPEntry)
    guarantees exists (e : RPEntry)
      (e in this.HasRPEntry.c and
       (this.authorized = true and e.rol = entry.rol and e.priv = entry.priv) or
       (this.authorized = false and (e.rol /= entry.rol or e.priv /= entry.priv)))

  dynamic model is
    event Auto();
    event DeletePriv(entry : in RPEntry);
    event PermDenied();
    event PermGranted();
    event UpdatePriv(entry : in RPEntry);
    event VerifyPriv(entry : in RPEntry);

    state Start;
    state Idle;
    state Verified;

    transition table is
      in Start on Auto to Idle;
      in Idle on DeletePriv do deletePriv to Idle;
      in Idle on UpdatePriv do updatePriv to Idle;
      in Idle on VerifyPriv do verifyPriv to Verified;
      in Verified on Auto if authorized = True send PermGranted to Idle;
      in Verified on Auto if authorized = False send PermDenied to Idle;
    end transition table;
  end dynamic model;
end class;

//*****
aggregation HasUsers is
  parent p : SecMgr multiplicity One;
  child c : SecSysUI multiplicity ZeroOrMore;
end aggregation;

aggregation HasAccMgr is
  parent p : SecMgr multiplicity One;
  child c : AccMgr multiplicity One;
end aggregation;

aggregation HasAdmMgr is
  parent p : SecMgr multiplicity One;
  child c : AdmMgr multiplicity One;
end aggregation;

aggregation HasAppMgr is
  parent p : SecMgr multiplicity One;
  child c : AppMgr multiplicity One;
end aggregation;

aggregation HasRepository is
  parent p : SecMgr multiplicity One;
  child c : Repository multiplicity One;
end aggregation;

```

```

aggregation HasLog is
  parent p : Repository multiplicity One;
  child c : Log multiplicity One;
end aggregation;

aggregation HasSession is
  parent p : Repository multiplicity One;
  child c : Session multiplicity One;
end aggregation;

aggregation HasRPTTable is
  parent p : Repository multiplicity One;
  child c : RPTTable multiplicity One;
end aggregation;

aggregation HasRPEntry is
  parent p : RPTTable multiplicity One;
  child c : RPEntry multiplicity ZeroOrMore;
end aggregation;

aggregation HasUATable is
  parent p : Repository multiplicity One;
  child c : UATable multiplicity One;
end aggregation;

aggregation HasUAEntry is
  parent p : UATable multiplicity One;
  child c : UAEntry multiplicity ZeroOrMore;
end aggregation;

aggregation HasUPTable is
  parent p : Repository multiplicity One;
  child c : UPTable multiplicity One;
end aggregation;

aggregation HasUPEntry is
  parent p : UPTable multiplicity One;
  child c : UPEntry multiplicity ZeroOrMore;
end aggregation;

aggregation HasURTable is
  parent p : Repository multiplicity One;
  child c : URTable multiplicity One;
end aggregation;

aggregation HasUREEntry is
  parent p : URTable multiplicity One;
  child c : UREEntry multiplicity ZeroOrMore;
end aggregation;

//*****
association NotAuthorized is
  role s1 : AppMgr multiplicity One;
  role r1 : SecSysUI multiplicity One;
end association;

association AckReq is
  role s1 : AppMgr multiplicity One;
  role r1 : SecSysUI multiplicity One;
end association;

association NoSession is
  role s1 : AppMgr multiplicity One;
  role r1 : SecSysUI multiplicity One;
end association;

association AppReq is
  role s1 : SecSysUI multiplicity One;
  role r1 : AppMgr multiplicity One;
end association;

```

```

association VerifySession is
    role s1 : AppMgr multiplicity One;
    role r1 : Session multiplicity One;
end association;

association InSession is
    role s1 : Session multiplicity One;
    role r1 : AppMgr multiplicity One;
end association;

association NotInSession is
    role s1 : Session multiplicity One;
    role r1 : AppMgr multiplicity One;
end association;

association LogEntry is
    role s1 : AppMgr multiplicity One;
    role s2 : AccMgr multiplicity One;
    role s3 : AdmMgr multiplicity One;
    role r1 : Log multiplicity One;
end association;

association GetRole is
    role s1 : AppMgr multiplicity One;
    role r1 : URTable multiplicity One;
end association;

association TheRole is
    role s1 : URTable multiplicity One;
    role r1 : AppMgr multiplicity One;
end association;

association VerifyPriv is
    role s1 : AppMgr multiplicity One;
    role r1 : RPTable multiplicity One;
end association;

association PermDenied is
    role s1 : RPTable multiplicity One;
    role r1 : AppMgr multiplicity One;
end association;

association PermGranted is
    role s1 : RPTable multiplicity One;
    role r1 : AppMgr multiplicity One;
end association;

association TheAttempts is
    role s1 : UATable multiplicity One;
    role r1 : AccMgr multiplicity One;
end association;

association GetAttempts is
    role s1 : AccMgr multiplicity One;
    role r1 : UATable multiplicity One;
end association;

association UpdateAttempts is
    role s1 : AccMgr multiplicity One;
    role r1 : UATable multiplicity One;
end association;

association GetPwd is
    role s1 : AccMgr multiplicity One;
    role r1 : SecSysUI multiplicity One;
end association;

association GrantedLogIn is
    role s1 : AccMgr multiplicity One;
    role r1 : SecSysUI multiplicity One;
end association;

```

```

association ThePwd is
    role s1 : SecSysUI multiplicity One;
    role r1 : AccMgr multiplicity One;
end association;

association InvalidLogIn is
    role s1 : AccMgr multiplicity One;
    role r1 : SecSysUI multiplicity One;
end association;

association Locked is
    role s1 : AccMgr multiplicity One;
    role r1 : SecSysUI multiplicity One;
end association;

association ReqLogIn is
    role s1 : SecSysUI multiplicity One;
    role r1 : AccMgr multiplicity One;
end association;

association ReqLogOut is
    role s1 : SecSysUI multiplicity One;
    role r1 : AccMgr multiplicity One;
end association;

association PwdFailed is
    role s1 : UPTable multiplicity One;
    role r1 : AccMgr multiplicity One;
end association;

association PwdPassed is
    role s1 : UPTable multiplicity One;
    role r1 : AccMgr multiplicity One;
end association;

association VerifyPwd is
    role s1 : AccMgr multiplicity One;
    role r1 : UPTable multiplicity One;
end association;

association UpdateSession is
    role s1 : AccMgr multiplicity One;
    role r1 : Session multiplicity One;
end association;

association UnLocked is
    role s1 : AdmMgr multiplicity One;
    role r1 : SecSysUI multiplicity One;
end association;

association DeletePriv is
    role s1 : AdmMgr multiplicity One;
    role r1 : RPTTable multiplicity One;
end association;

association UpdatePriv is
    role s1 : AdmMgr multiplicity One;
    role r1 : RPTTable multiplicity One;
end association;

association DeletePwd is
    role s1 : AdmMgr multiplicity One;
    role r1 : UPTable multiplicity One;
end association;

association UpdatePwd is
    role s1 : AdmMgr multiplicity One;
    role r1 : UPTable multiplicity One;
end association;

```

```

association DeleteRole is
    role s1 : AdmMgr multiplicity One;
    role r1 : URTTable multiplicity One;
end association;

association UpdateRole is
    role s1 : AdmMgr multiplicity One;
    role r1 : URTTable multiplicity One;
end association;

association ResetLog is
    role s1 : AdmMgr multiplicity One;
    role r1 : Log multiplicity One;
end association;

association ViewLog is
    role s1 : AdmMgr multiplicity One;
    role r1 : Log multiplicity One;
end association;

association DeleteSession is
    role s1 : AccMgr multiplicity One;
    role r1 : Session multiplicity One;
end association;

end package;

```

Appendix E. Channel Domain Model (AWL)

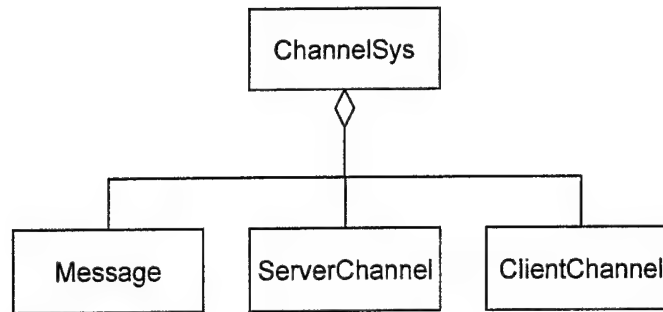


Figure 75. Channel Aggregate Domain Model

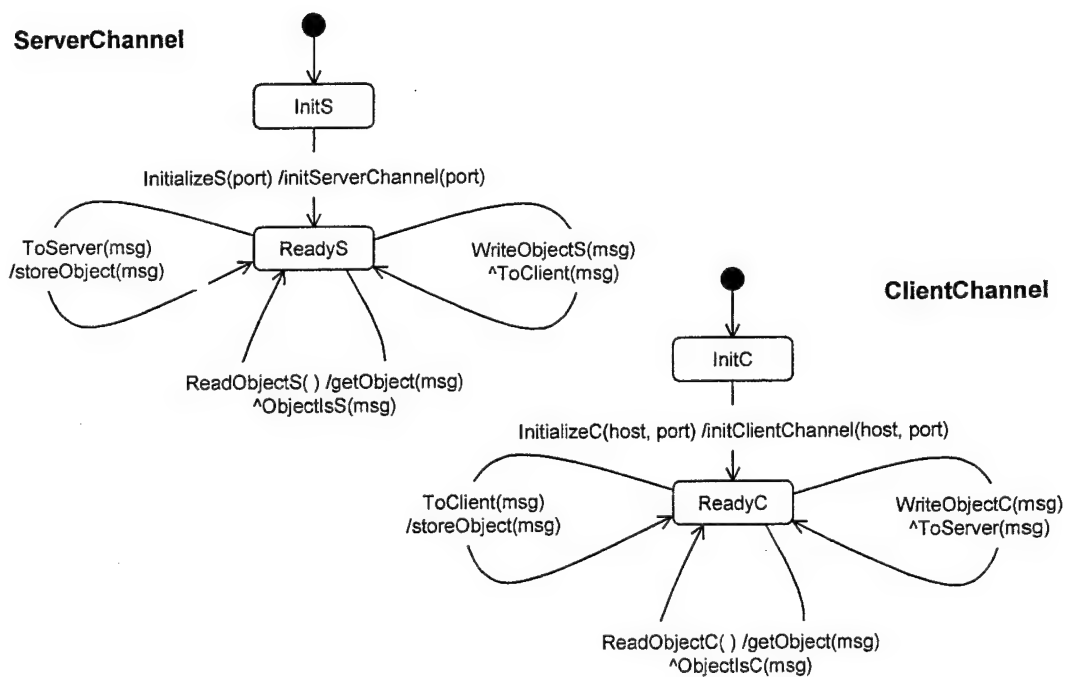


Figure 76. Channel's Client and Server Dynamic Models

```

/* THIS SOFTWARE AND ANY ACCOMPANYING DOCUMENTATION IS RELEASED "AS IS."
* THE U.S. GOVERNMENT MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
* CONCERNING THIS SOFTWARE AND ANY ACCOMPANYING DOCUMENTATION, INCLUDING,
* WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A
* PARTICULAR PURPOSE. IN NO EVENT WILL THE U.S. GOVERNMENT BE LIABLE
* FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER
* INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE, OR
* INABILITY TO USE, THIS SOFTWARE OR ANY ACCOMPANYING DOCUMENTATION,
* EVEN IF INFORMED IN ADVANCE OF THE POSSIBILITY OF SUCH DAMAGES.*/

```

```

/*****
Channel Domain Model (Channel.awl)

```

Description:

The Channel Domain model was developed to satisfy the need to have a generic high-level analysis domain model to represent multi-agent systems. Channel models the communication that occurs between a client and a server in a distributed agent system. The communication between the two types of agents pass a message object that represents any/all communication required by the application being modeled.

Interface Contract:

Event	Inter/Intra Parameters	Class Parameter Type	Description
InitializeC	inter-model	ClientChannel (receiver)	
	host	STRING	
	port	NATURAL	
Initializes	inter-model	ServerChannel (receiver)	
	port	NATURAL	
ObjectIsC	inter-model	ClientChannel (sender)	"returns" messages from the client.
	msg	Message	
ObjectIsS	inter-model	ServerChannel (sender)	"returns" messages from the server.
	msg	Message	
ReadObjectC	inter-model	ClientChannel (receiver)	
ReadObjectS	inter-model	ServerChannel (receiver)	
ToServer	intra-model		Communication from the client to the server.
	msg	Message	
ToClient	intra-model		Communication from the server to the client.
	msg	Message	
WriteObjectC	inter-model	ClientChannel (receiver)	"feeds" messages to the client.
	msg	Message	
WriteObjectS	inter-model	ServerChannel (receiver)	"feeds" messages to the server.
	msg	Message	
	rwc	RoomWithCapy	

History:

Original - (01/29/01) Create by Dr. Hartrum.
 01/29/01 - Updated to reflect the "well-formed" domain rules. (Nonnweiler)
 02/13/01 - Fixed in/out modes for the client and server events (Nonnweiler)

```

*****/

```

package Channel is

```

type OBJECT is abstract;
type CHAR is abstract;
type STRING is sequence of CHAR;
type NATURAL is range 0 .. *;
type INTEGER is range * .. *;
type One is range 1 .. 1;
type ZeroOrMore is range 0 .. *;

```



```

//-----
class ChannelSys is
end class;

//-----
class Message is

  /*****
  * The Message class is the unit of information physically sent between agents. It
  * is a passive class, with no state model and no methods except the implicit Gets
  * and Sets. This is completely predefined (implemented) in Java in agentMom. When
  * sent, the sending agent fills in the sender, host, and port attributes, along
  * with the performative, content, and any other attributes as needed. The
  * attributes are based on KQML:
  *
  * host          - The sender's host name.
  * port          - The sender's port number.
  * sender        - The message originator's identification.
  * receiver      - The message recipient's identification.
  * performative  - The message's purpose; basically the message type.
  * force         - For future use (see KQML).
  * inreplyto     - For future use (see KQML).
  * language      - For future use (see KQML).
  * ontology      - For future use (see KQML).
  * replywith     - For future use (see KQML).
  * content       - An object containing the information.
  *****/

  public host          : STRING;
  public port          : NATURAL;
  public sender        : STRING;
  public receiver      : STRING;
  public performative  : STRING;
  public force         : STRING;
  public inreplyto     : STRING;
  public language      : STRING;
  public ontology      : STRING;
  public replywith     : STRING;
  public content       : OBJECT;

  public procedure InitMessage()
    guarantees
      host' = []
      and port' = 0
      and sender' = []
      and receiver' = []
      and performative' = []
      and force' = []
      and inreplyto' = []
      and language' = []
      and ontology' = []
      and replywith' = []
      and content' = null
  end class;

type MessageQueue is sequence of Message;

function queue(pos : in NATURAL): Message
  guarantees true // TBD later

```

```

//-----
class ClientChannel is

  /*****
  * The ClientChannel is a
  *
  * host - Host name of the target ServerChannel.
  * portNo - Port number of the target ServerChannel.
  *****/

  public host : STRING;
  public portNo : NATURAL;
  public queue : MessageQueue;

  public procedure initClientChannel(host : in STRING, port : in NATURAL)
    guarantees portNo' = port and this.host' = host

  public procedure storeObject(msg : in Message)
    guarantees msg in queue'

  public procedure getObject(msg : out Message)
    guarantees msg' = queue(1) and msg' in queue'

  dynamic model is
    event AUTO();
    event InitializeC(host : in STRING, port : in NATURAL);
    event WriteObjectC(msg: in Message);
    event ReadObjectC();
    event ObjectIsC(msg: out Message);
    event ToServer(msg: out Message);
    event ToClient(msg: in Message);

    state START;
    state InitC;
    state ReadyC;

    transition table is
      in START on AUTO to InitC;
      in InitC on InitializeC do initClientChannel to ReadyC;
      in ReadyC on WriteObjectC send ToServer to ReadyC;
      in ReadyC on ReadObjectC do getObject send ObjectIsC to ReadyC;
      in ReadyC on ToClient do storeObject to ReadyC;
    end transition table;
  end dynamic model;
end class;

//-----
class ServerChannel is

  /*****
  * The ServerChannel is a
  *
  * localPort - Local port number to listen on.
  *****/

  public localPort : NATURAL;
  public queue : MessageQueue;

  public procedure initServerChannel(port : in NATURAL)
    guarantees LocalPort' = port

  public procedure storeObject(msg : in Message)
    guarantees msg in queue'

  public procedure getObject(msg : out Message)
    guarantees msg' = queue(1) and msg' in queue'

```

```

dynamic model is
  event AUTO();
  event InitializeS(port : in NATURAL);
  event WriteObjectS(msg: in Message);
  event ReadObjectS();
  event ObjectIsS(msg: out Message);
  event ToServer(msg: in Message);
  event ToClient(msg: out Message);

  state START;
  state InitS;
  state ReadyS;

  transition table is
    in START on AUTO to InitS;
    in InitS on InitializeS do initServerChannel to ReadyS;
    in ReadyS on WriteObjectS send ToClient to ReadyS;
    in ReadyS on ReadObjectS do getObject send ObjectIsS to ReadyS;
    in ReadyS on ToServer do storeObject to ReadyS;
  end transition table;
end dynamic model;
end class;

//-----
association ToServer is
  role s : ClientChannel multiplicity One;
  role r : ServerChannel multiplicity One;
end association ;

association ToClient is
  role s : ServerChannel multiplicity One;
  role r : ClientChannel multiplicity One;
end association ;

//-----
aggregation HasMessage is
  parent p : ChannelsSys multiplicity One;
  child c : Message multiplicity ZeroOrMore;
end aggregation;

aggregation HasClientChannel is
  parent p : ChannelsSys multiplicity One;
  child c : ClientChannel multiplicity ZeroOrMore;
end aggregation;

aggregation HasServerChannel is
  parent p : ChannelsSys multiplicity One;
  child c : ServerChannel multiplicity ZeroOrMore;
end aggregation;
end package;

```

```

/*****
* The following code is the resulting MICs created from the integration with the Room
* Manager Domain Model. The entire model is not shown as it is easily derived from the
* listed models.
*****/

```

```

//-----
class MICCP1 is
  public triggerTS1 : BOOLEAN;
  public r_GetRoom : RoomContainer;

  public function EvaluateTS1() : BOOLEAN
    guarantees EvaluateTS1 = TRUE
  public procedure ProcessGetRoom(r : in Room)
    guarantees exists (x1 : Room)
      ((x1 in r_GetRoom) and (x1 = r) and (triggerTS1' = EvaluateTS1()))
  public function ConvertmsgWriteObjectC() : Message
    guarantees exists (m : Message, r : Room)
      ((r in r_GetRoom) and (m.content = r) and (ConvertmsgWriteObjectC = m))
  public procedure ConversionTS1(msg : out Message)
    guarantees msg = ConvertmsgWriteObjectC()
  public procedure InitializeTS1()
    guarantees (r_GetRoom' = {}) and (triggerTS1' = FALSE)

  dynamic model is
    event AUTO();
    event GetRoom(r : in Room);
    event WriteObjectC(msg : out Message);

    state START;
    state ReceiveEventsTS1;
    state ConverteddedTS1;

    transition table is
      in START on AUTO do InitializeTS1 to RecieveEventsTS1;
      in ReceiveEventsTS1 on GetRoom do ProcessGetRoom to RecieveEventsTS1;
      in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1
        send WriteObjectC to ConverteddedTS1;
      in ConverteddedTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
    end transition table;
  end dynamic model;
end class;

```

```

//-----
class MICCP2 is
  public triggerTS1 : BOOLEAN;
  public msg_ObjectIsS : MessageContainer;

  public function EvaluateTS1() : BOOLEAN
    guarantees EvaluateTS1 = TRUE
  public procedure ProcessObjectIsS(msg : in Message)
    guarantees exists (x1 : Message)
      ((x1 in msg_ObjectIsS) and (x1 = msg) and (triggerTS1' = EvaluateTS1()))
  public function ConverterGetRoom() : Room
    guarantees exists (m : Message, r : Room)
      ((m in msg_ObjectIsS) and (r = m.content) and (ConverterGetRoom = r))
  public procedure ConversionTS1(r : out Room)
    guarantees r = ConverterGetRoom()
  public procedure InitializeTS1()
    guarantees (msg_ObjectIsS' = {}) and (triggerTS1' = FALSE)

```

```

dynamic model is
  event AUTO();
  event ObjectIsS(msg : in Message);
  event GetRoom(r : out Room);

  state START;
  state ReceiveEventsTS1;
  state ConvertededTS1;

  transition table is
    in START on AUTO do InitializeTS1 to RecieveEventsTS1;
    in ReceiveEventsTS1 on ObjectIsS do ProcessObjectIsS to RecieveEventsTS1;
    in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1
      send GetRoom to ConvertededTS1;
    in ConvertededTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
  end transition table;
end dynamic model;
end class;

//-----
class MICCP3 is
  public triggerTS1 : BOOLEAN;
  public c_GetRWC : NATURALContainer;

  public function EvaluateTS1() : BOOLEAN
    guarantees EvaluateTS1 = TRUE
  public procedure ProcessGetRWC(c : in NATURAL)
    guarantees exists (x1 : NATURAL)
      ((x1 in c_GetRWC) and (x1 = c) and (triggerTS1' = EvaluateTS1()))
  public function ConvertmsgWriteObjectC() : Message
    guarantees exists (m : Message, n : NATURAL)
      ((n in c_GetRWC) and (m.content = n) and (ConvertmsgWriteObjectC = m))
  public procedure ConversionTS1(msg : out Message)
    guarantees msg = ConvertmsgWriteObjectC()
  public procedure InitializeTS1()
    guarantees (c_GetRWC' = {}) and (triggerTS1' = FALSE)

  dynamic model is
    event AUTO();
    event GetRWC(c : in NATURAL);
    event WriteObjectC(msg : out Message);

    state START;
    state ReceiveEventsTS1;
    state ConvertededTS1;

    transition table is
      in START on AUTO do InitializeTS1 to RecieveEventsTS1;
      in ReceiveEventsTS1 on GetRWC do ProcessGetRWC to RecieveEventsTS1;
      in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1
        send WriteObjectC to ConvertededTS1;
      in ConvertededTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
    end transition table;
  end dynamic model;
end class;

//-----
class MICCP4 is
  public triggerTS1 : BOOLEAN;
  public msg_ObjectIsS : MessageContainer;

  public function EvaluateTS1() : BOOLEAN
    guarantees EvaluateTS1 = TRUE
  public procedure ProcessObjectIsS(msg : in Message)
    guarantees exists (x1 : Message)
      ((x1 in msg_ObjectIsS) and (x1 = msg) and (triggerTS1' = EvaluateTS1()))
  public function ConvertcGetRWC() : NATURAL
    guarantees exists (m : Message, n : NATURAL)
      ((m in msg_ObjectIsS) and (n = m.content) and (ConvertcGetRWC = n))
  public procedure ConversionTS1(c : out NATURAL)
    guarantees c = ConvertcGetRWC()

```

```

public procedure InitializeTS1()
    guarantees (msg_ObjectIsS' = {}) and (triggerTS1' = FALSE)

dynamic model is
    event AUTO();
    event ObjectIsS(msg : in Message);
    event GetRWC(c : out NATURAL);

    state START;
    state ReceiveEventsTS1;
    state ConvertededTS1;

    transition table is
        in START on AUTO do InitializeTS1 to RecieveEventsTS1;
        in ReceiveEventsTS1 on ObjectIsS do ProcessObjectIsS to RecieveEventsTS1;
        in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1
            send GetRWC to ConvertededTS1;
        in ConvertededTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
    end transition table;
end dynamic model;
end class;

//-----
class MICCP5 is
    public triggerTS1 : BOOLEAN;
    public rwc_NewRWC : RoomWithCapyContainer;

    public function EvaluateTS1() : BOOLEAN
        guarantees EvaluateTS1 = TRUE
    public procedure ProcessNewRWC(rwc : in RoomWithCapy)
        guarantees exists (x1 : RoomWithCapy)
            ((x1 in rwc_NewRWC) and (x1 = rwc) and (triggerTS1' = EvaluateTS1()))
    public function ConvertmsgWriteObjectC() : Message
        guarantees exists (m : Message, rwc : RoomWithCapy)
            ((rwc in rwc_NewRWC) and (m.content = rwc) and (ConvertmsgWriteObjectC = m))
    public procedure ConversionTS1(msg : out Message)
        guarantees msg = ConvertmsgWriteObjectC()
    public procedure InitializeTS1()
        guarantees (rwc_NewRWC' = {}) and (triggerTS1' = FALSE)

dynamic model is
    event AUTO();
    event NewRWC(rwc : in RoomWithCapy);
    event WriteObjectC(msg : out Message);

    state START;
    state ReceiveEventsTS1;
    state ConvertededTS1;

    transition table is
        in START on AUTO do InitializeTS1 to RecieveEventsTS1;
        in ReceiveEventsTS1 on NewRWC do ProcessNewRWC to RecieveEventsTS1;
        in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1
            send WriteObjectC to ConvertededTS1;
        in ConvertededTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
    end transition table;
end dynamic model;
end class;

//-----
class MICCP6 is
    public triggerTS1 : BOOLEAN;
    public msg_ObjectIsS : MessageContainer;

    public function EvaluateTS1() : BOOLEAN
        guarantees EvaluateTS1 = TRUE
    public procedure ProcessObjectIsS(msg : in Message)
        guarantees exists (x1 : Message)
            ((x1 in msg_ObjectIsS) and (x1 = msg) and (triggerTS1' = EvaluateTS1()))
    public function ConverttrwcNewRWC() : RoomWithCapy
        guarantees exists (m : Message, rwc : RoomWithCapy)

```

```

    ((m in msg_ObjectIsS) and (rwc = m.content) and (ConverttrwcNewRWC = rwc))
public procedure ConversionTS1(rwc : out RoomWithCapy)
    guarantees rwc = ConverttrwcNewRWC()
public procedure InitializeTS1()
    guarantees (msg_ObjectIsS' = {}) and (triggerTS1' = FALSE)

dynamic model is
    event AUTO();
    event ObjectIsS(msg : in Message);
    event NewRWC(rwc : out RoomWithCapy);

    state START;
    state ReceiveEventsTS1;
    state ConverteddedTS1;

    transition table is
        in START on AUTO do InitializeTS1 to RecieveEventsTS1;
        in ReceiveEventsTS1 on ObjectIsS do ProcessObjectIsS to RecieveEventsTS1;
        in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1
            send NewRWC to ConverteddedTS1;
        in ConverteddedTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
    end transition table;
end dynamic model;
end class;

//-----
class MICCP7 is
    public triggerTS1 : BOOLEAN;
    public msg_ObjectIsC : MessageContainer;
    public function EvaluateTS1() : BOOLEAN
        guarantees EvaluateTS1 = TRUE
    public procedure ProcessObjectIsC(msg : in Message)
        guarantees exists (x1 : Message)
            ((x1 in msg_ObjectIsC) and (x1 = msg) and (triggerTS1' = EvaluateTS1()))
    public function ConverttrwcRWC() : RoomWithCapy
        guarantees exists (m : Message, rwc : RoomWithCapy)
            ((m in msg_ObjectIsC) and (rwc = m.content) and (ConverttrwcRWC = rwc))
    public procedure ConversionTS1(rwc : out RoomWithCapy)
        guarantees rwc = ConverttrwcRWC()
    public procedure InitializeTS1()
        guarantees (msg_ObjectIsC' = {}) and (triggerTS1' = FALSE)

dynamic model is
    event AUTO();
    event ObjectIsC(msg : in Message);
    event RWC(rwc : out RoomWithCapy);

    state START;
    state ReceiveEventsTS1;
    state ConverteddedTS1;

    transition table is
        in START on AUTO do InitializeTS1 to RecieveEventsTS1;
        in ReceiveEventsTS1 on ObjectIsC do ProcessObjectIsC to RecieveEventsTS1;
        in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1
            send RWC to ConverteddedTS1;
        in ConverteddedTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
    end transition table;
end dynamic model;
end class;

```

```

//-----
class MICCP8 is
  public triggerTS1 : BOOLEAN;
  public rwc_RWC : RoomWithCapyContainer;

  public function EvaluateTS1() : BOOLEAN
    guarantees EvaluateTS1 = TRUE
  public procedure ProcessRWC(rwc : in RoomWithCapy)
    guarantees exists (x1 : RoomWithCapy)
      ((x1 in rwc_RWC) and (x1 = rwc) and (triggerTS1' = EvaluateTS1()))
  public function ConvertmsgWriteObjectS() : Message
    guarantees exists (m : Message, rwc : RoomWithCapy)
      ((rwc in rwc_RWC) and (m.content = rwc) and (ConvertmsgWriteObjectS = m))
  public procedure ConversionTS1(msg : out Message)
    guarantees msg = ConvertmsgWriteObjectS()
  public procedure InitializeTS1()
    guarantees (rwc_RWC' = {}) and (triggerTS1' = FALSE)

  dynamic model is
    event AUTO();
    event RWC(rwc : in RoomWithCapy);
    event WriteObjectS(msg : out Message);

    state START;
    state ReceiveEventsTS1;
    state ConvertededTS1;

    transition table is
      in START on AUTO do InitializeTS1 to RecieveEventsTS1;
      in ReceiveEventsTS1 on RWC do ProcessRWC to RecieveEventsTS1;
      in ReceiveEventsTS1 on AUTO if triggerTS1 = TRUE do ConversionTS1
        send WriteObjectS to ConvertededTS1;
      in ConvertededTS1 on AUTO do InitializeTS1 to ReceiveEventsTS1;
    end transition table;
  end dynamic model;
end class;

```


Appendix F. ADMIT Configuration Management

Note: both systems (the verifier and the integrator) require the following packages:

- AWSOME.WsClasses
- AWSOME.WsIntegration
- AWSOME.Parser
- AWSOME.Verifier
- java.io.*
- java.util.*
- java.awt.*
- javax.swing.*

Well-Formed Domain Model Verifier:

/ AWSOME / testdrivers / **ModelVerifierDriver** (*executable*)
/ AWSOME / testdrivers / ModelVerifierPane

AWSOME Domain Model Integration Tool:

/ AWSOME / WsIntegration / **ADMIT** (*executable*)
/ AWSOME / WsIntegration / AWLGenerator
/ AWSOME / WsIntegration / CommPattern
/ AWSOME / WsIntegration / CounterTS
/ AWSOME / WsIntegration / EventTS
/ AWSOME / WsIntegration / ModelBuilder
/ AWSOME / WsIntegration / ModelDeconflictor
/ AWSOME / WsIntegration / ModelEntry
/ AWSOME / WsIntegration / ModelEntryPane
/ AWSOME / WsIntegration / ReceiveEvent
/ AWSOME / WsIntegration / SendEvent
/ AWSOME / WsIntegration / TriggerStrategy
/ AWSOME / WsIntegration / ValueTS

Available AWL domain models:

- RoomSys.awl (*Room Manager System*)
- SecSys.awl (*Security Manager System*)
- demo1.awl (*Test cases*)
- demo2.awl (*Test cases*)
- model1.awl (*Test cases*)
- model2.awl (*Test cases*)
- model3.awl (*Test cases*)

Bibliography

1. Batory & O'Malley. "The Design and Implementation of Hierarchical Software Systems With Reusable Components". University of Texas, Department of Computer Sciences, Austin, TX
2. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. New York: ACM Press, 1999
3. David Garlan, Mary Shaw. "An Introduction to Software Architecture". School of Computer Science. Carnegie Mellon University Pittsburgh, PA 15213.
4. David Harel, Eran Gery. "Executable Object Modeling with Statecharts". *IEEE*, 31-42 (July 1997)
5. Gary Anderson. *An Interactive Tool for Refining Software Specifications from a Formal Domain Model*. MS thesis, Air Force Institute of Technology, WPAFB, OH, March 1999. AFIT/GCS/ENG/99M-01. ADA361745.
6. James Odell and others. "Extending UML for Agents" James Odell Associates, Ann Arbor, MI 48103.
7. Markus Kaiser. "Carolina Version x.x: Reference & Documentation", (First Draft). University of Connecticut.
8. Mitchell Saba and Eugene Santos. "The Multi-Agent Distributed Goal Satisfaction System". University of Connecticut, Intelligent Distributed Information Systems Lab.
9. Pierre-Alain Muller. *Instant UML*. Wrox Press Ltd. Paris, France. 1997
10. Robert Graham. Class handout, CSCE 793, Advanced Topics in Software Engineering. Air Force Institute of Technology, Department of Electrical and Computer Engineering WPAFB, OH.
11. Robert Graham. Lecture, CSCE 793, Advanced Topics in Software Engineering. Air Force Institute of Technology, Department of Electrical and Computer Engineering WPAFB, OH.
12. Roger S. Pressman. *Software Engineering: A Practitioner's Approach*, (Fourth Edition). New York: McGraw-Hill, 1997
13. Rumbaugh and others. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1991.
14. Scott A. DeLoach. "Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems". *Agent-Oriented Information Systems (AOIS)*. Seattle Washington: May 1999.

15. Stephen Garland, Nancy Lynch. "Using I/O Automata for Developing Distributed Systems". MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.
16. Thomas Hartrum. "Formal Synthesis of *agentMom* Systems". Air Force Institute of Technology, Department of Electrical and Computer Engineering WPAFB, OH.
17. Thomas Hartrum, Robert Graham, Jr. "AFIT Knowledge-Based Software Engineering (KBSE) Research Status Report – June 1999". Air Force Institute of Technology, Department of Electrical and Computer Engineering WPAFB, OH.
18. Thomas Hartrum, Robert Graham, Jr. "The AFIT Wide Spectrum Object Modeling Environment: An AWSOME Beginning". Air Force Institute of Technology, Department of Electrical and Computer Engineering WPAFB, OH.
19. Thomas Hartrum, Timothy Karagias. "Generation of Object-Oriented Formal Software Specifications". Air Force Institute of Technology, Department of Electrical and Computer Engineering WPAFB, OH. *Proceedings of the IEEE 1997 National Aerospace and Electronics Conference (NAECON)*. Dayton OH: July 1997.

Vita

Joel Carl Nonnweiler was born in Buena Park, California in September 1966, and graduated from Eureka High School, California in June 1985. He married Brenda Gallaty in June 1987, and enlisted in the Air Force in February 1988. He earned an Associate in Arts degree from the University of Maryland, European Division, in June 1994. After attaining the rank of Staff Sergeant, he separated from active duty to enter the Air Force Reserve Officer Training Commissioning program, Detachment 470. He was commissioned after graduating with a Bachelor of Science degree in Computer Science from the University of Nebraska at Omaha in December 1997. His first officer assignment was to the 4th Space Operations Squadron, Schriever AFB, CO where he worked with the communication apportionment for the MILSTAR satellite communication system. Joel and Brenda have two children, their son Andrew and daughter Nicole.